Processes

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

- Multi-programming vs. Multi-tasking
- Program vs. Process
- Forking
- External Programs
- More PCB
- Process Management & Scheduling
- Context switching
- Unix Signals
- Re-entrant Functions
- Signal Handling
- CPU Utilization
- Process Creation

Multi-Programming and Multi-Tasking

Multiprogramming and multitasking

- early computers had limited memory
- only one program could run at a time
- lengthy I/O → idle CPU

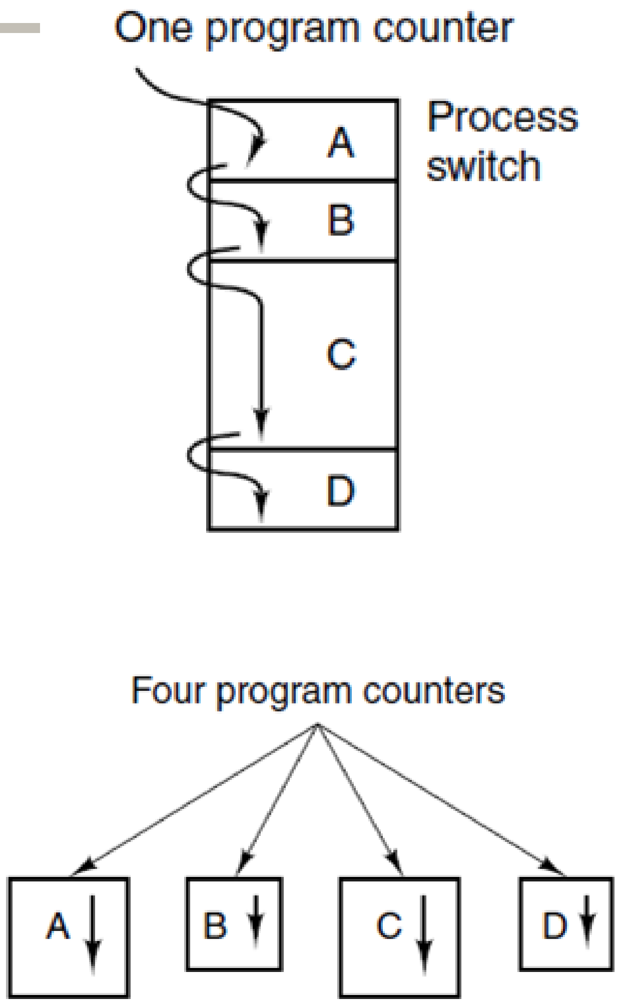
- cheaper memory ⟶ multiple programs can be loaded simultaneously
- **multiprogramming** – OS gives CPU to another program if current program must wait on I/O

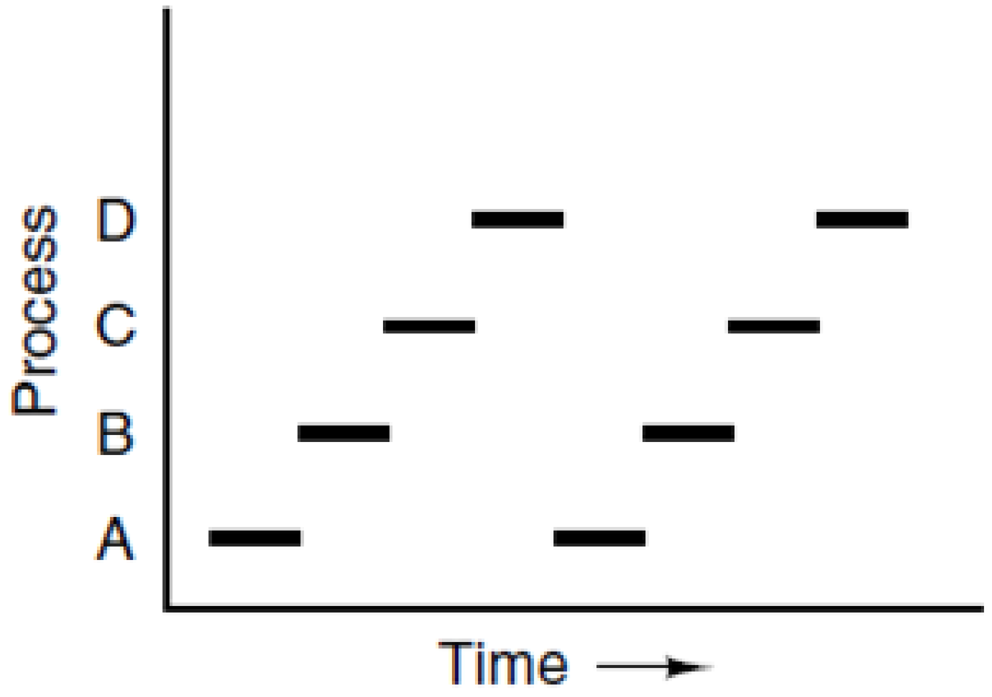
- **cooperative multitasking** - programs can voluntarily yield CPU to another program
- early Windows and Mac OS supported this
- you can manually yield using `sleep(0)` or `sched_yield()`

- **preemptive multitasking** – a program gets a fraction of a second to execute, then OS automatically switches to the next program, and so on
- nearly all modern OSes implement this

- **multithreading** – allows even more efficient multitasking, usually preemptive

Multiprogramming on a single CPU





Multitasking

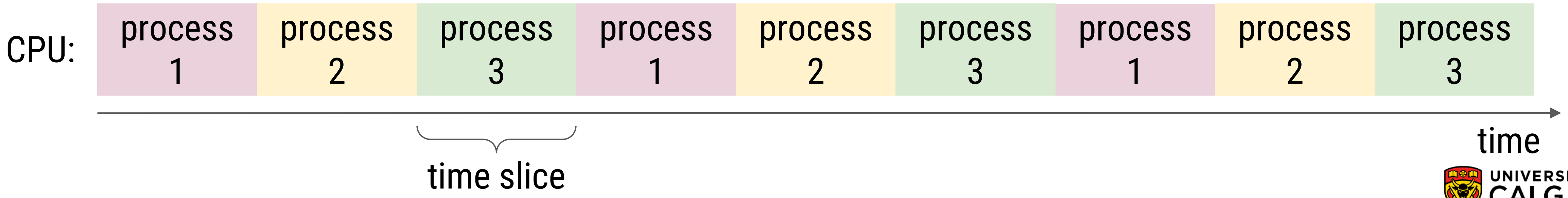
- Multi-tasking allows a computer with **M** CPUs to run **N** processes, even when **M < N**
- Multi-tasking = concurrent* execution of multiple programs
- just an illusion of parallelism – programs do not actually have to execute their instructions simultaneously, as long as it appears that way
- just like multi-programming, multi-tasking also reduces CPU idling during I/O

Multitasking

- imagine a single CPU running multiple programs:

```
repeat forever:
  for every process p:
    give CPU to process p for a short time (fraction of a second)
```

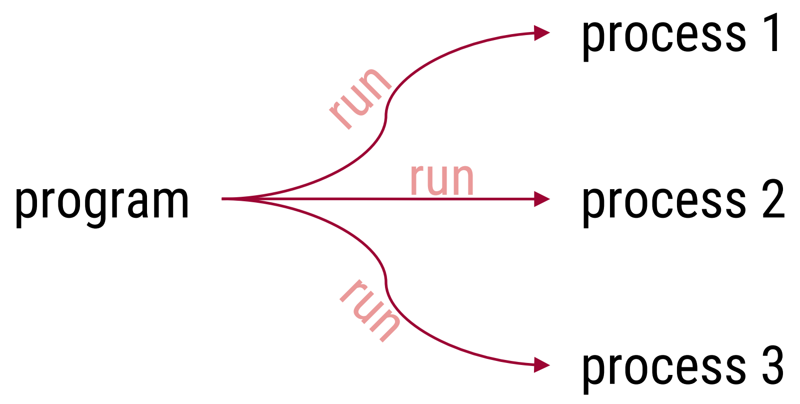
- if process decide what "short time" is, we have cooperative multi-tasking
- if OS decided what "short time" is, we have pre-emptive multi-tasking
  - short time is called time slice, or quantum



Program vs. Process

Program vs Process

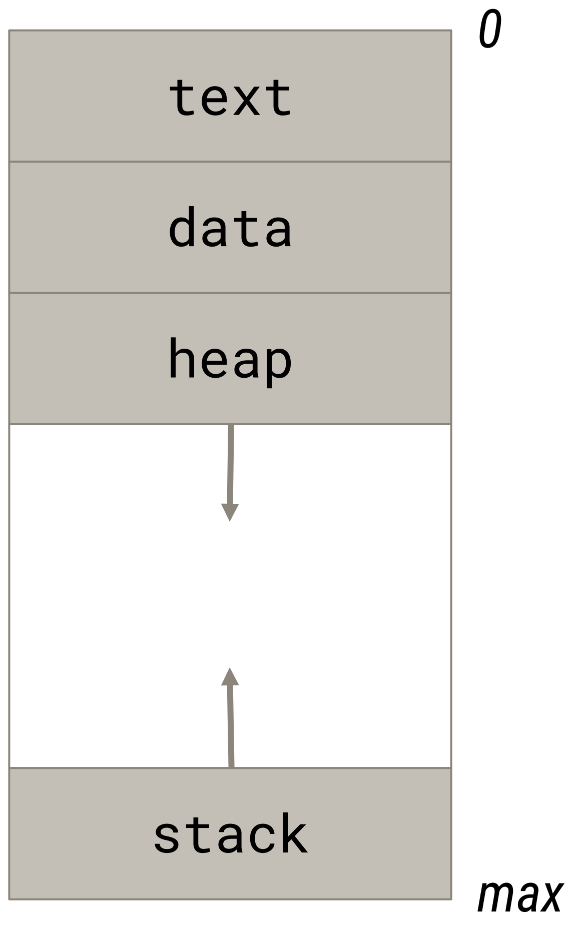
- OS needs to keep track of all running programs
- a program is a passive entity – executable file containing a list of instructions, e.g. stored on disk
- a **process** is an active entity – e.g. with a unique program counter, saved registers, open files
- a program becomes a process when it starts to be loaded into memory for execution
- a single program can be used to start multiple processes
  e.g. running multiple terminals or shells



A process in memory

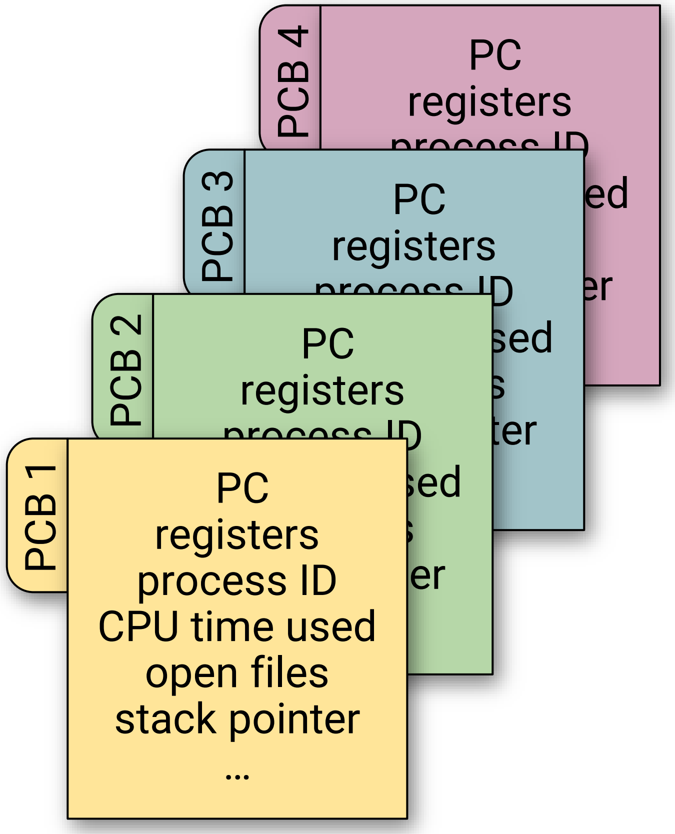
- each process gets its own address space
  - part(s) of memory available to a process, decided by OS
  - on modern OSes it is a *virtual* address space (0 … max), isolated from other processes

- examples of things in address space of a process:
  - text section: the program code
  - data section: global variables
  - heap: dynamically allocated memory
  - stack: parameters, return address, local variables



Process control block (PCB)

- OSes track many types of information to run a process
- all this data is stored somewhere, in some data structure
- we will call such data structure a Process Control Block (PCB)
- on Linux, PCB is called `task_struct`
- typical parts of PCB:
  - program counter + other CPU registers
  - process state
  - process priority
  - memory management info: e.g. page table
  - accounting info: e.g. CPU time, process number
  - I/O status info: e.g. open files
- a process table is a collection of all PCBs (e.g. array, linked list, etc)



Operations on processes

- OS allows processes to be created/deleted dynamically
- process creation in UNIX is accomplished by calling the `fork()` system call
    - process that calls `fork()` is called the parent process
    - the newly created process is called the child process
    - processes in the system form a process tree
    - each process gets PID - a unique process identifier, usually an `unsigned int`
- process execution, e.g. calling `fork()` in Unix
- process termination, e.g. calling `exit()` or `kill()`
    - ask OS to delete the process and free up resources
    - termination can be only requested by the process itself, its parent, or an unrelated process provided its owner has adequate permissions

Multi-process

A multiprocess program in C

```
$ man fork

pid_t fork(void);

fork()  creates  a new process by duplicating the calling process.  The
new process is referred to as the child process.  The  calling  process
is referred to as the parent process.

The child process and the parent process run in separate memory spaces.
At the time of fork() both memory spaces have the same content.  Memory
writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed
by one of the processes do not affect the other.

The child process is an exact duplicate of the  parent  process  except
for the following points:
...
```

A multiprocess program in C

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

Possible output:

???

A multiprocess program in C

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

Possible output:

```
Hello
world.
world.
```

Are other outputs possible?

A multiprocess program in C

https://repl.it/@jonathanwhudson/fork-hello-world
https://repl.it/@jonathanwhudson/fork-hello-world-one-char-at-a-time

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

Possible output:

```
Hello
world.
world.
```

Another possible output:

```
Hello
worwold.
rld.
```

assuming `printf()` is unbuffered, printing 1 character at a time

Are other outputs possible?

A multiprocess program in C

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

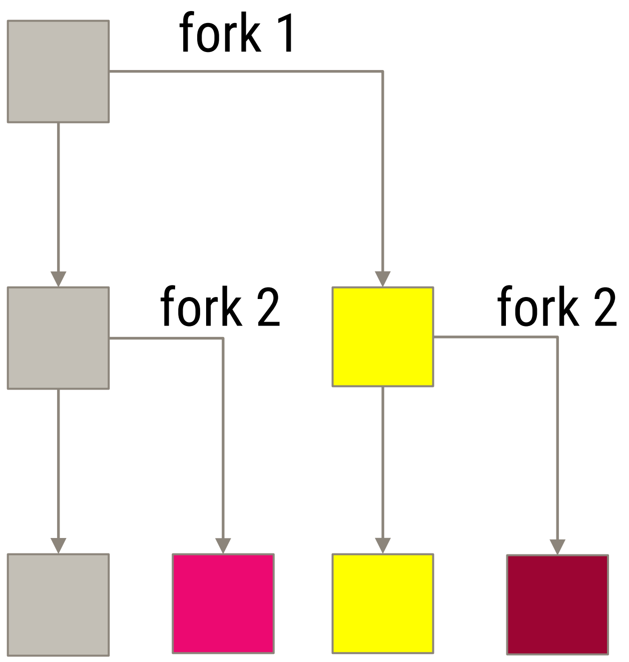
If `fork()` fails, it returns **-1**. You should always check the return value of a system call.

Possible output:
```
Hello
world.
world.
```

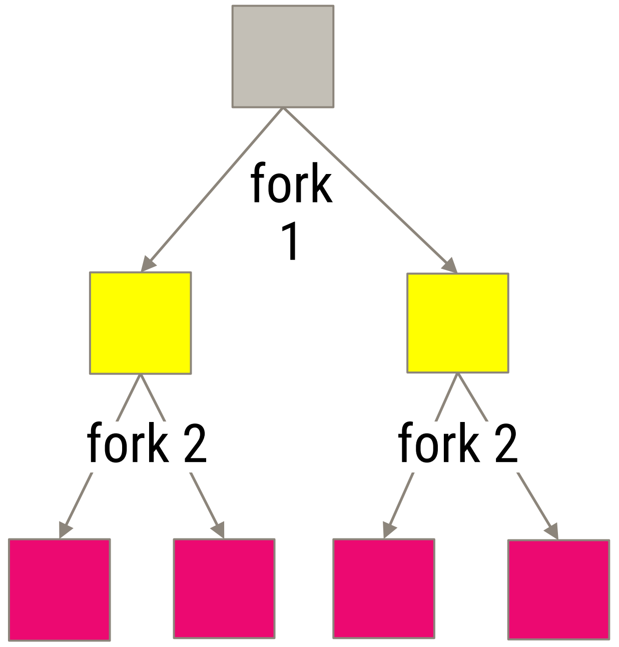
Another possible output:
```
Hello
worwold.
rld.
```

Another possible output:
```
Hello
world.
```

2 different ways to think about fork()



```
int main() {
    fork(); //1
    fork(); //2
}
```



A multiprocess program in C

```
int main() {
    /* remember the return value */
    pid_t pid = fork();
    /* both parent and child will execute the
next line,
     * but will have different value for pid:
     *       0 for child,
     *      >0 (positive) for parent,
     *      -1 for error */
    printf("fork returned %d.\n", pid);
}
```

Possible output:

```
fork returned 7.
fork returned 0.
```

Possible output:

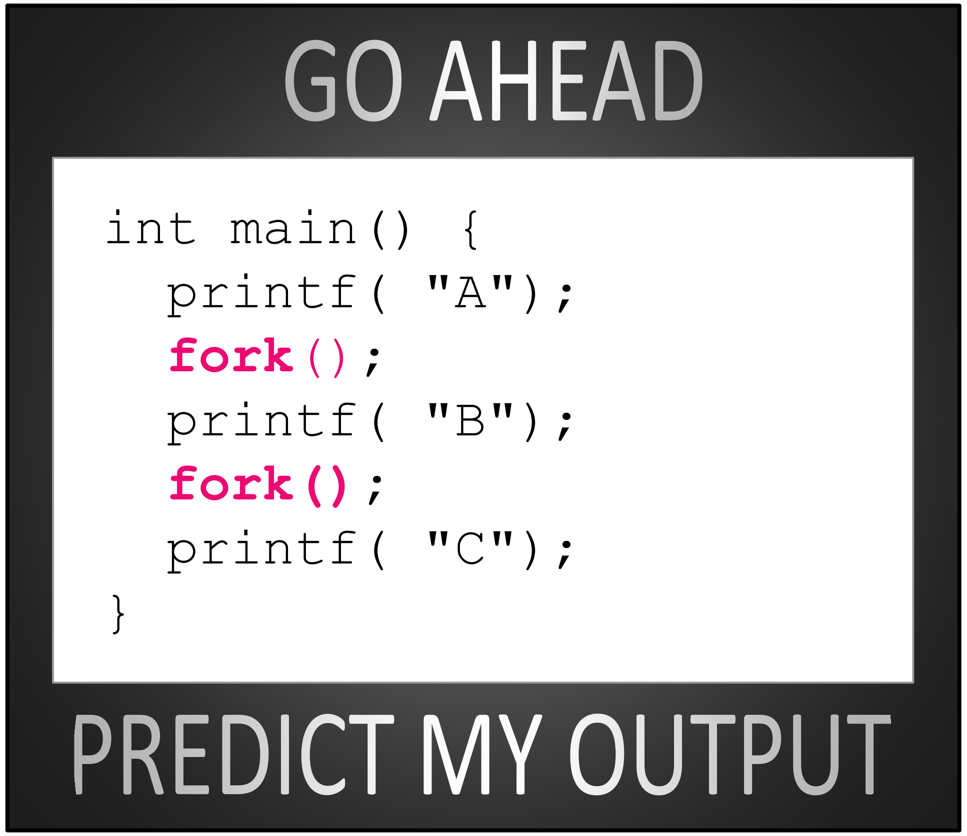
```
fork returned 0.
fork returned 7198.
```

Possible output:

```
fork returned -1.
```

Forking Exercise

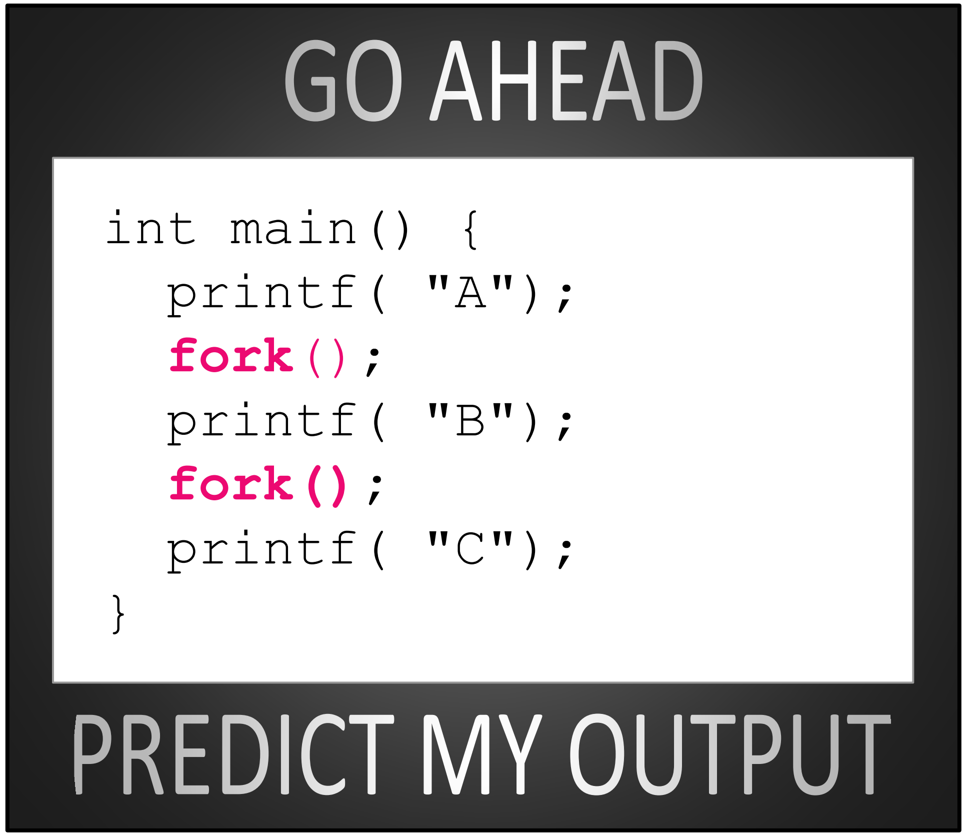
Exercise


GO AHEAD
PREDICT MY OUTPUT

```
int main() {
  printf( "A");
  fork();
  printf( "B");
  fork();
  printf( "C");
}
```

Can you predict all possible outputs assuming fork() does not fail?

Exercise – helps you to stay in shape

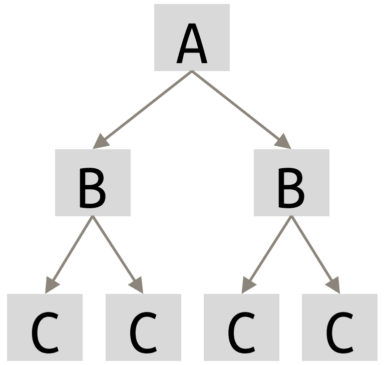

GO AHEAD

```
int main() {
  printf( "A");
  fork();
  printf( "B");
  fork();
  printf( "C");
}
```

PREDICT MY OUTPUT

Hint:

- find all unique topological orderings in the graph



- e.g. ABCCBCC

Another forking exercise – predict all outputs

```
int main() {
    for(int i=0 ; i<4 ; i++ ) {
        fork();
    }
    printf("X");
}
```

Easy version:
assume fork()
does not fail

Harder version:
assume fork()
could fail

Another forking exercise – predict all outputs

```
int main() {
    for(int i=0 ; i<4 ; i++ ) {
        fork();
    }
    printf("X");
}
```

Hint: is this program equivalent to the one on the left?

```
int main()
{
    fork();
    fork();
    fork();
    fork();
    printf("X");
}
```

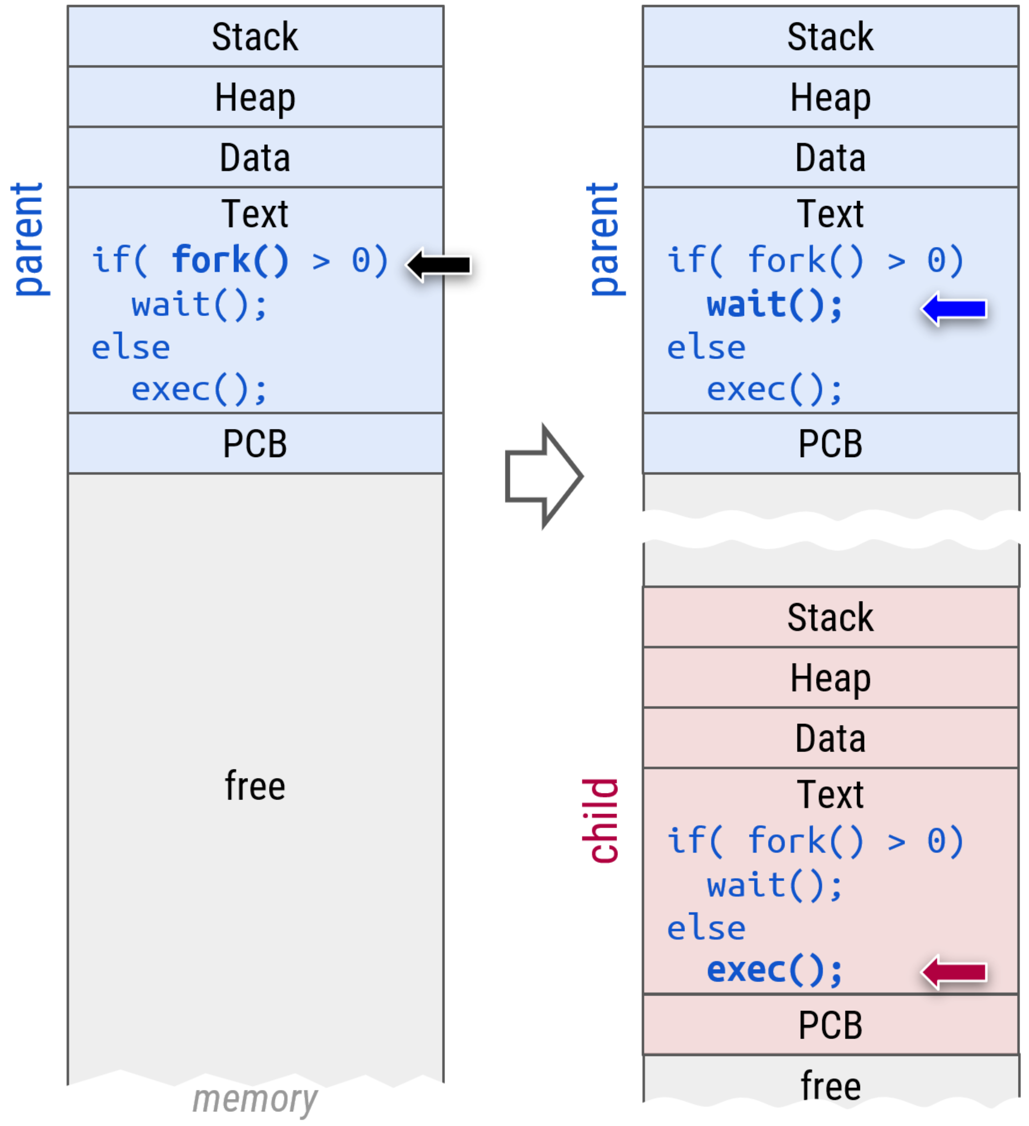
Another forking exercise – predict all outputs

```
int main() {
    for(int i=0 ; i<4 ; i++ ) {
        fork();
        printf("%d",i);
    }
}
```

HInt: This is actually very similar to the first exercise...

Address space & fork

- `fork()` duplicates address space, creating nearly identical copy of itself
- the next instruction is the same for both parent and child, although it is usually "`if-else`", so code flow afterwards is typically different for child vs. parent



Exercise – can you predict the output?

```
int x = 10;

int main() {
    printf("x=%d\n", x);
    fork();
    x ++;
    printf("x=%d\n", x);
}

// assume fork() does not
fail
```

Hint:
child has its own 'x' variable, different from the parent

Fork bomb

WARNING

Do not run this on CPSC machines.

```
int main() {
    while(1) {
        fork();
    }
    printf("X"); //
???
}
```

External Programs

UNIVERSITY OF CALGARY

Starting an external program (programmatically)

- how do we start an external program in Unix?
- no dedicated system call for this purpose
- we have to `fork()` a new process
- then we replace child process with an external program using `exec()` system call

Starting an external program (programmatically)

```
$ man execve

int execve(const char *filename, char *const argv[], char *const envp[]);

execve() executes the program pointed to by filename.  filename must be
either a binary executable, or a script starting with  a  line  of  the
form:
           #! interpreter [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is  an  array  of argument strings passed to the new program.  By
convention, the first of these  strings  should  contain  the  filename
associated  with the file being executed.  envp is an array of strings,
conventionally of the form key=value, which are passed  as  environment
to  the  new  program.  Both argv and envp must be terminated by a null
pointer.

...

execve() does not return on success, and the text, initialized data,
uninitialized data (bss), and stack of the calling process are
overwritten according to the contents of the newly loaded program.

...
```

Starting an external program (programmatically)

```
$ man execlp

int execlp(const char *file, const char *arg, ...);

The  exec() family of functions replaces the current process image with
a new process image.  The functions described in this manual  page  are
front-ends  for execve(2).  (See the manual page for execve(2) for fur-
ther details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that
to be executed.

...
```

execlp() is easier to use than execve() for most people

Starting external program with fork() + exec()

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main() {
    pid_t pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process created successfully */
        /* replace process with 'ls -l' */
        execlp("/bin/ls", "ls", "-l", NULL);
        /* execlp only returns if it fails, in which case errno contains the reason */
        printf("execlp failed\n");
        _exit(-1); /* _exit() is normally recommended for children */
    }
    else { /* parent process will wait for the child to complete */
        printf("Waiting for child process %d\n", pid);
        while (wait(NULL) > 0) {;} /* in this case, wait(NULL) would also work */
        printf("Child finished.\n");
        exit(0);
    }
}
```

A multiprocess program in C

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("Before ls.\n");
  system("/bin/ls -l");
  printf("After ls.\n");
}
```

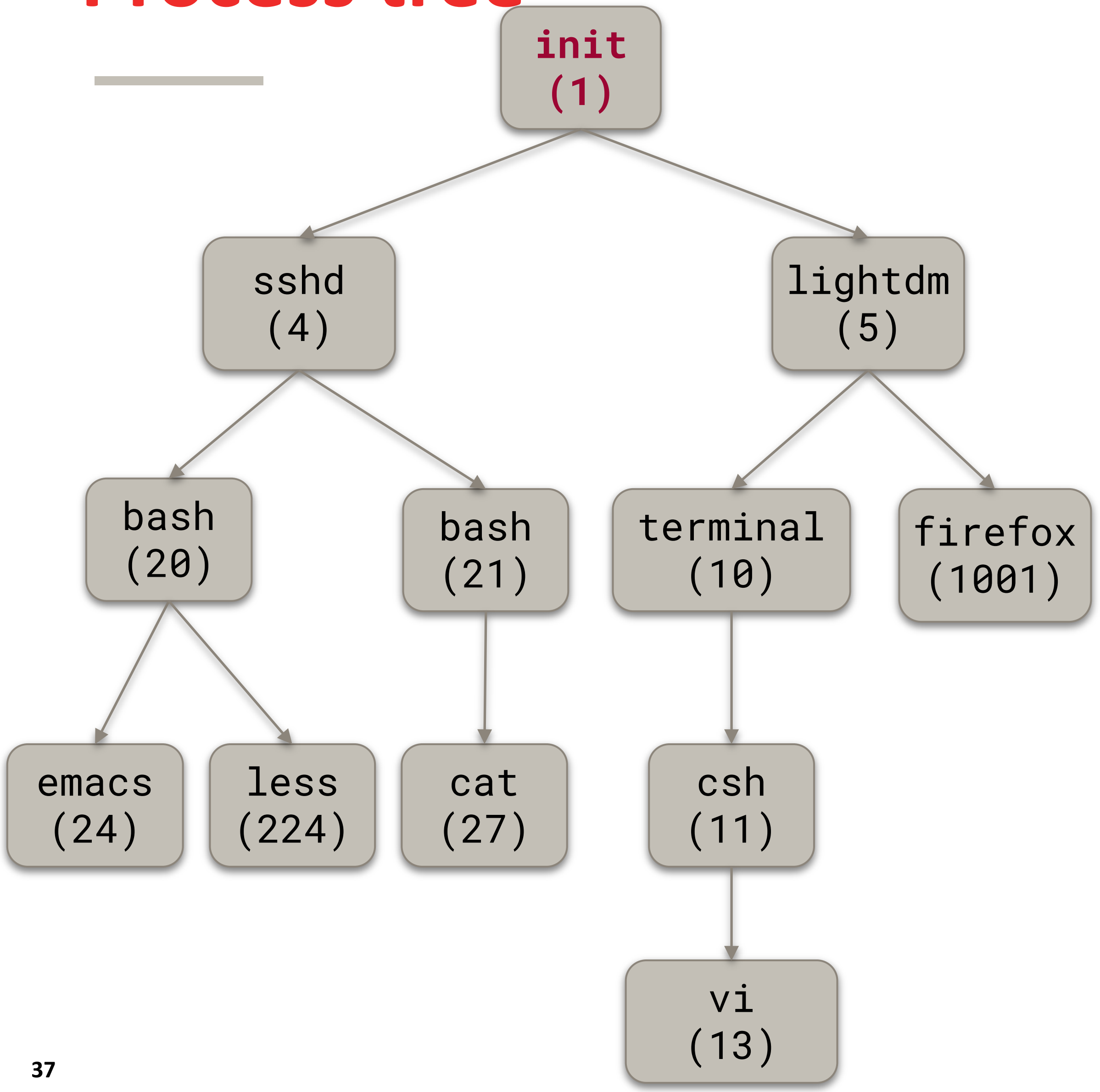
```
$ man system
...
The system() library function uses fork(2)
to create a child process that executes the
shell command specified in command using
execl(3) as follows:

    execl("/bin/sh", "sh", "-c", command,
          (char *) 0);

system() returns after the command has been
completed.
...
```

A multiprocess program in C

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE * fp = popen("/bin/ls -l", "r");
  if (fp == NULL) {
    fprintf( stderr, "popen failed.\n");
    exit(-1);
  }

  char buff[4096];
  while (fgets(buff, sizeof(buff), fp) != NULL)
    printf("%s", buff);

  pclose(fp);
}
```

```
$ man popen
...

The  popen()  function opens a
process by creating a pipe,
forking, and invoking the shell.
Since a pipe is by definition
unidirectional, the type argument
may specify only reading or
writing,  not  both;  the
resulting stream is
correspondingly read-only or
write-only.
```

Process tree



- **parent process** - the creating process
- **child process (child)** - the newly created process
- **PID** - the unique process identifier for each process

- in Unix, parent and child processes continue to be associated, forming a process hierarchy

- in Windows, all processes are equal, the parent process can give the control of its children to any other process

init process

- `init` or `systemd` is the first process started after booting
  - older UNIX systems used `init`
  - many/most newer Linux distros switched to `systemd`
- `init` is the ancestor of all user processes (direct or indirect parent), i.e. root of process tree
- `init` always has PID = 1
- orphaned processes are adopted by `init` (parent terminates before child)
- printing a process tree

```
$ pstree
$ ps axjf
```

- note: some special 'system processes' are created by kernel during boot, and do not have to be descendants of init, such as `swapper` and `pagedaemon`

More PCB

UNIVERSITY OF CALGARY

More examples of fields of a PCB

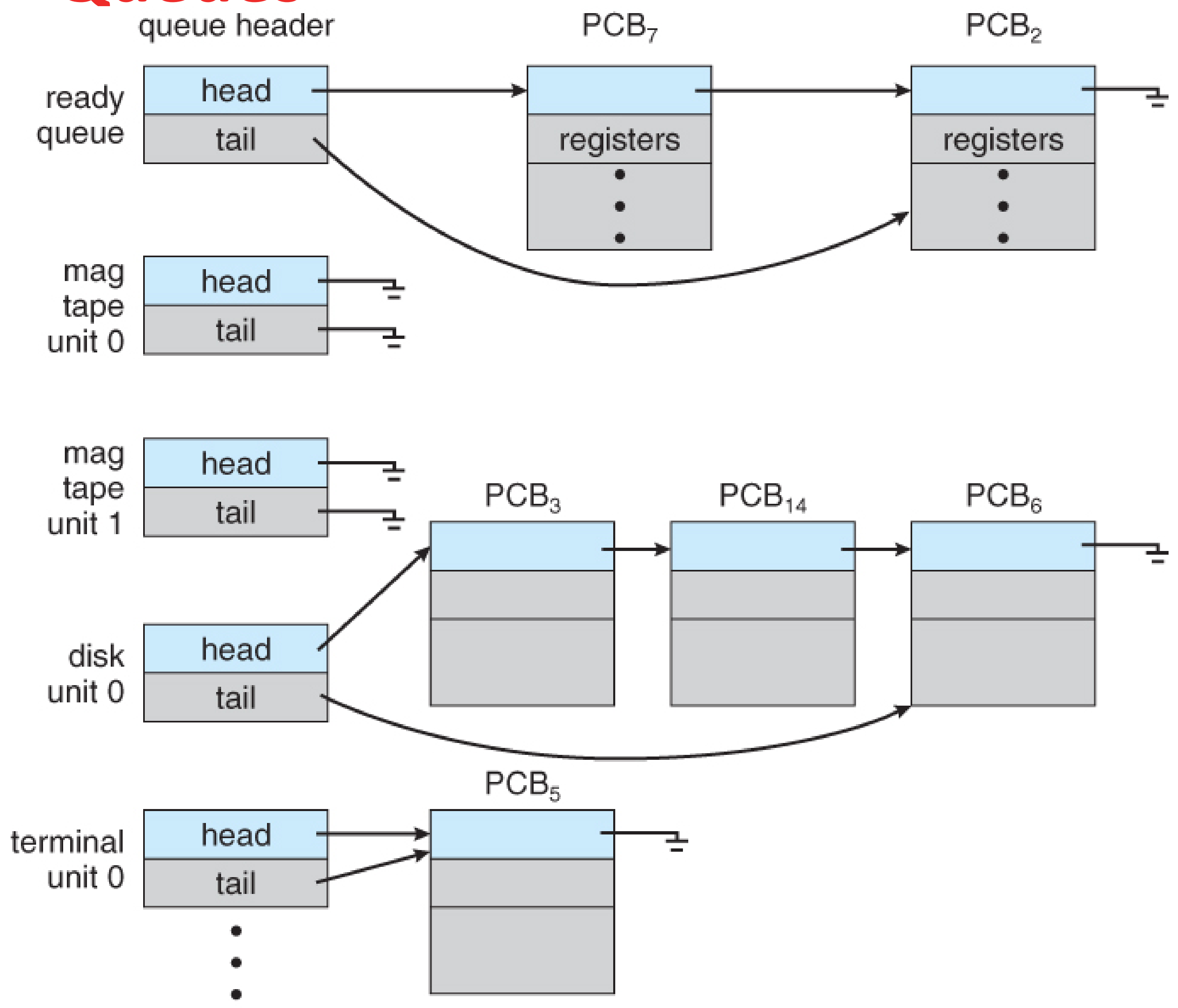
Process management	**Memory management**	**File management**
program counter	pointer to text segment	root directory
registers	pointer to data segment	working directory
stack pointer	pointer to stack segment	file descriptors
process state	...	user ID
priority		group ID
scheduling parameters		...
process ID		
parent process		
process group		
signals		
process start time		
CPU time used		
children's CPU time used		
time of next alarm		
...		

Look for "`task_struct`" in Linux kernel sources

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

UNIVERSITY OF CALGARY

Queues



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

Process Management

Common parent-child execution scenarios

after child process is created, parent process usually does one of 3 things:

- **parent waits** until child is finished,
  often used when child executes another program
- e.g. `fork+exec+wait()`, or `system()`

```
pid = fork()
if pid > 0 :
    wait()
```

- **parent continues** to execute concurrently and **independently** of the child,
- e.g. browser launches PDF viewer after downloading PDF

```
pid = fork()
if pid > 0 :
    do_whatever()
    exit()
```

- **parent continues** to execute concurrently with child, and periodically **synchronizes** with the child
- e.g. `popen()`
- we won't cover this scenario, because we will instead focus on threads and related synchronization mechanisms

```
pid = fork()
if pid > 0 :
    do_something_1()
    synchronize()
    do_something_2()
    synchronize()
    ...
```

Process termination

- voluntary:
    - normal exit - eg. application decides to terminate, or user instructs an app to 'close'
        - app calls `exit(0)` or returns `0` from `main()` – which is the same thing in C
    - error exit - application detects an error, optionally notifies user, and then terminates
        - app calls `exit(N)` or returns `N` from `main()` with `N!=0`
- involuntary:
    - fatal error – aka bugs in software
        - error detected by OS, e.g. accessing invalid memory, division by zero
    - external – killed by another process
        - parent, or another process calls `kill()`
        - e.g. system shutdown, pressing <ctrl-c> in terminal, closing GUI window

Process termination

- parent may decide to terminate its children for different reasons, for example:
    - the task assigned to the child is no longer required
    - the parent needs/wants to exit and wants to clean up first
- what happens when parent process is terminated before child (UNIX):
    - the child processes may be terminated, or assigned to the grandparent process, or to the `init` process
    - process hierarchy is always maintained
    - default behavior on Linux is to reparent the child process to the `init` process
    - this can be changed (e.g. to kill children, reparent to some other process)
      see `prctl()` for more details

Process termination

- process termination is not cheap

- after process terminates, OS needs to clean up:
  - free memory used by the process
  - delete PCB
  - delete process from process table
  - kill children or assign them a new parent
  - close open files
  - close network connections

  . . .

Process Scheduling

Process scheduling (basics)

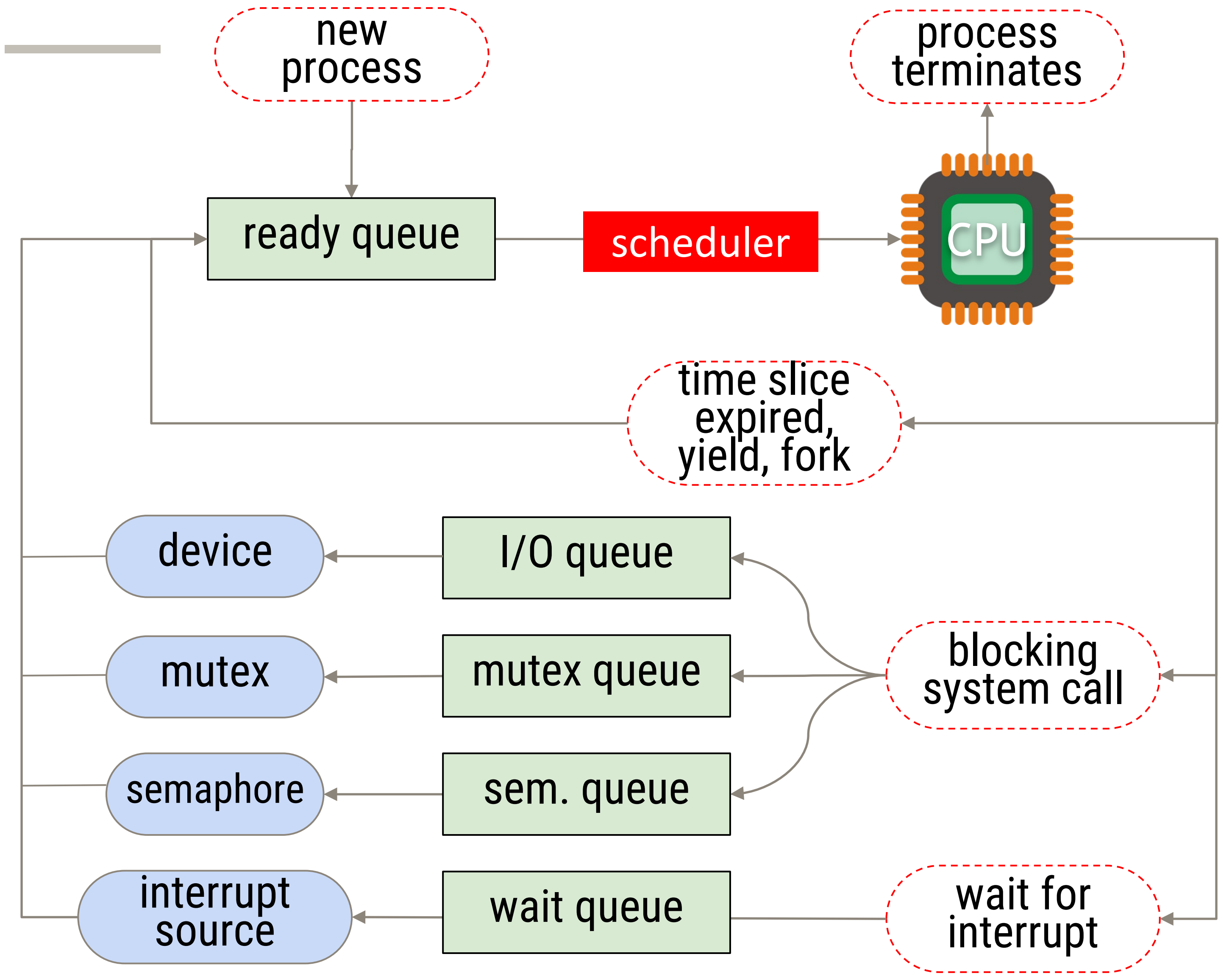
- part of multitasking is deciding which process gets the CPU next
- typical objective is to maximize CPU utilization

- **process scheduler**:
  - kernel routine/algorithm that chooses one of available process to execute next on the CPU
  - selected from processes in a ready queue
  - ready queue: all processes that are ready to execute their next instruction
    - e.g. linked list, priority queue, balanced binary search tree

- OS maintains other scheduling queues as well:
  - job queue: all programs waiting to run, usually found in batch systems
    - e.g. priority queue

  - device queues: processes waiting for a particular device
    - each device has its own queue

When OS needs to invoke scheduler



- variety of reasons
- basically any time CPU becomes available
- examples:
    - process yields
    - thread calls mutex lock
    - time slice expires

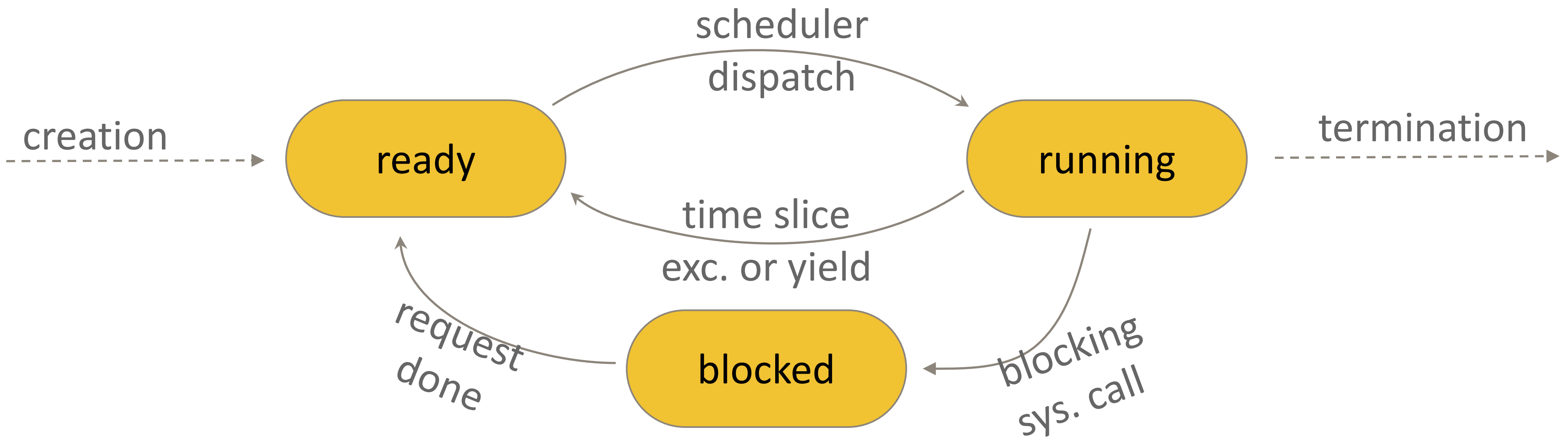
Process states

3 process states:

- **running** — actually running on the CPU
- **blocked** — waiting for some event to occur, eg. I/O
- **ready** — the process is ready to execute on CPU

only 4 transitions are possible:

- ready → running
- running → ready
- running → blocked
- blocked → ready



Process Scheduling Exercise

Exercise – simulating round-robin scheduling

- simulate 3 processes A, B, C
  - A: 7 units of CPU, 1 unit of I/O, 7 units of CPU
  - B: 4 CPU, 1 I/O, 3 CPU, 1 I/O, 1 CPU
  - C: 5 CPU
- assume time slice of 3 units
  - each process gets 3 units of CPU cycles
  - if process requests I/O during its time slice, OS switches to the next process
  - otherwise, after time slices expires, OS switches to next process
- assume I/O is very short, less than 1 time-slice

A	B	C
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	i/o	cpu
cpu	cpu	
cpu	cpu	
i/o	cpu	
cpu	i/o	
cpu	cpu	
cpu		
cpu		
cpu		
cpu		
cpu		

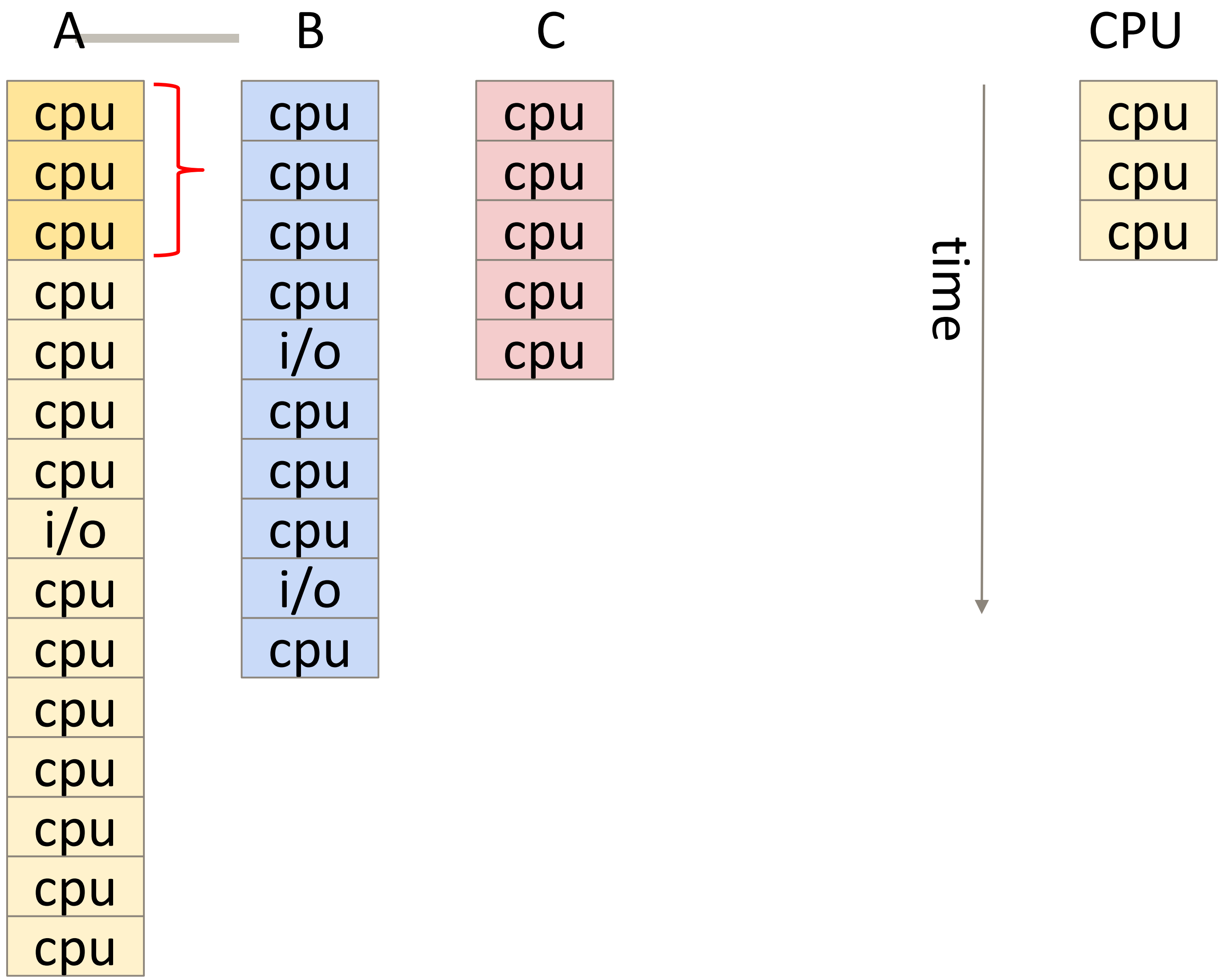
Exercise

A	B	C
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	i/o	cpu
cpu	cpu	
cpu	cpu	
i/o	cpu	
cpu	i/o	
cpu	cpu	
cpu		
cpu		
cpu		
cpu		
cpu		

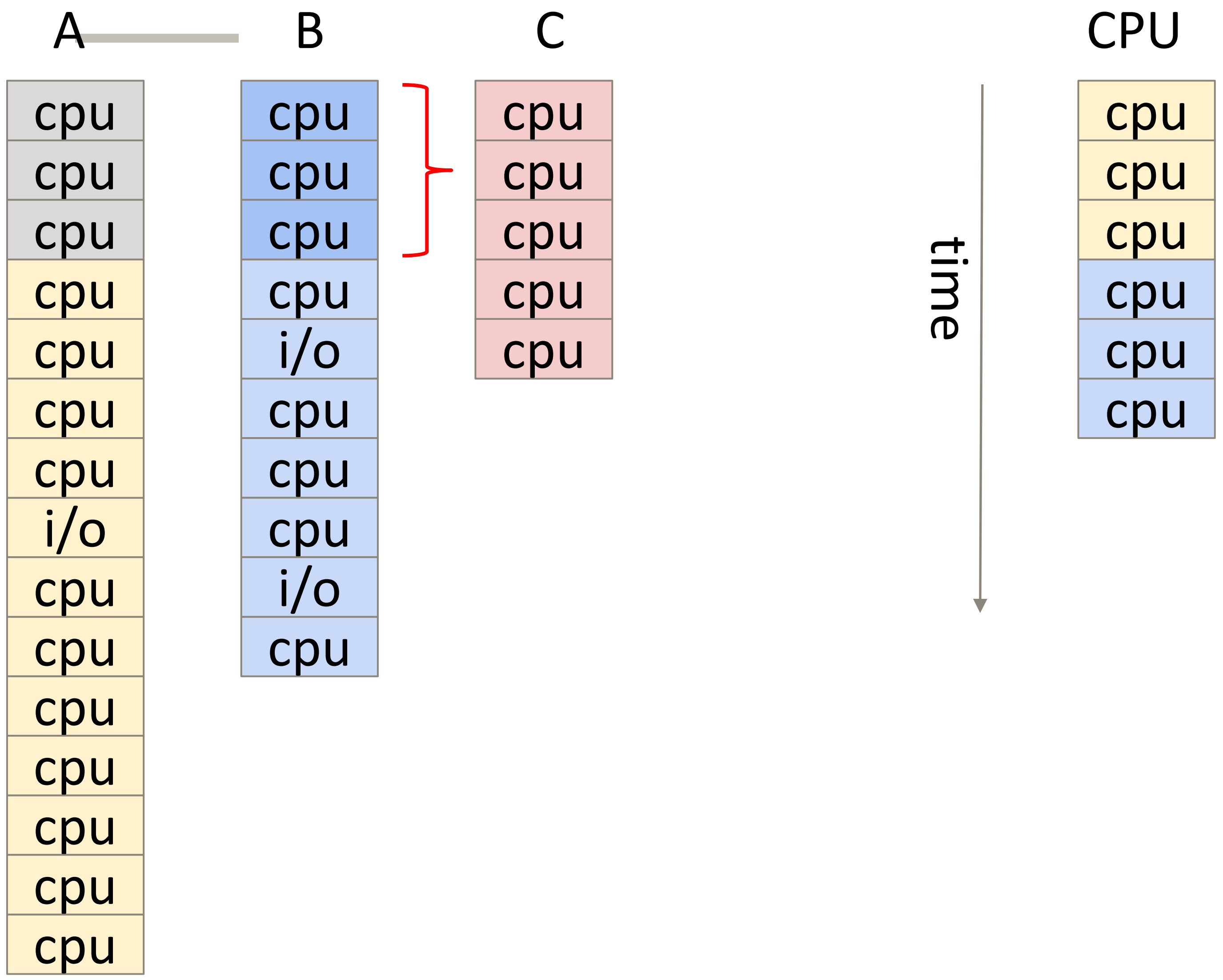
time

CPU

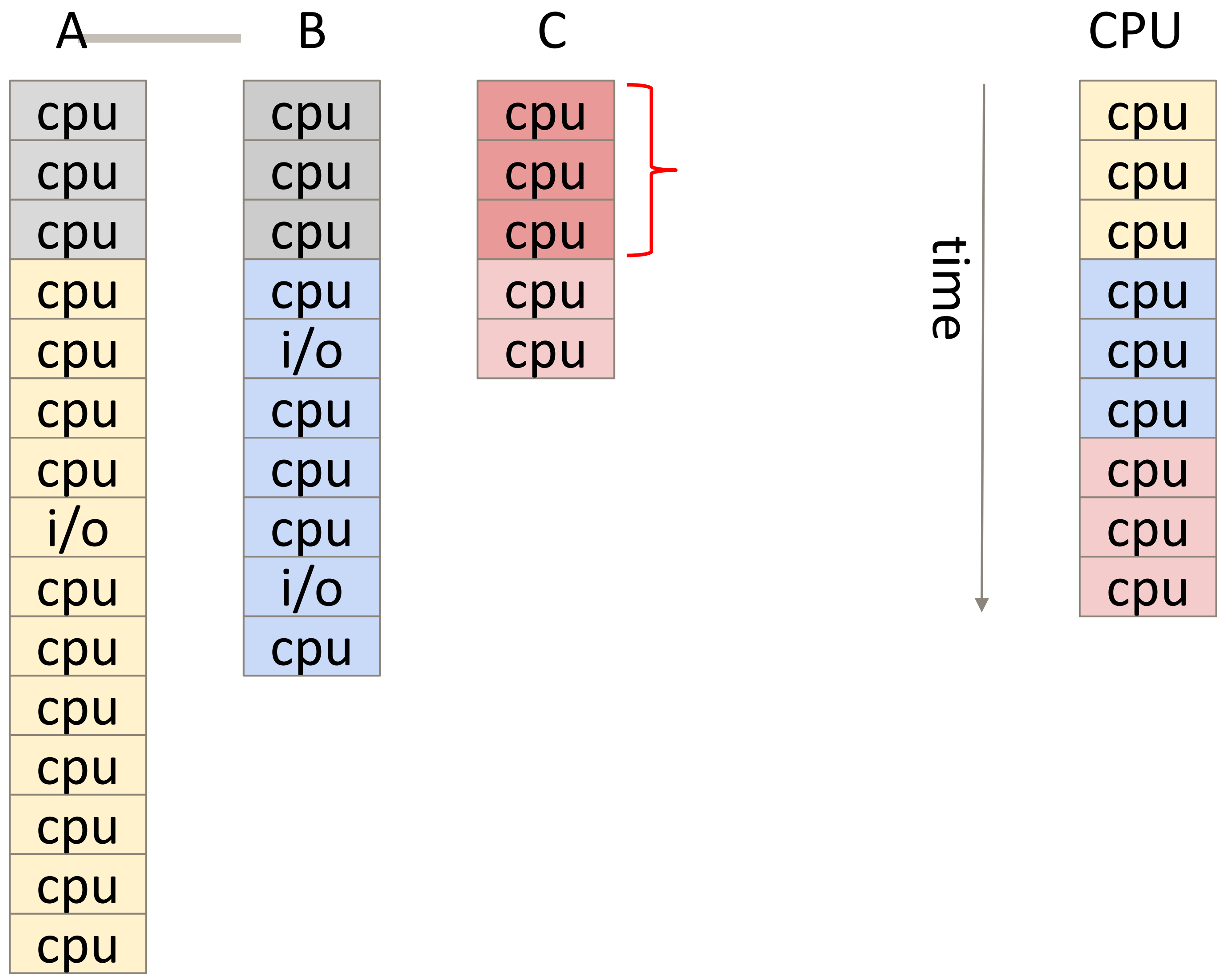
Exercise



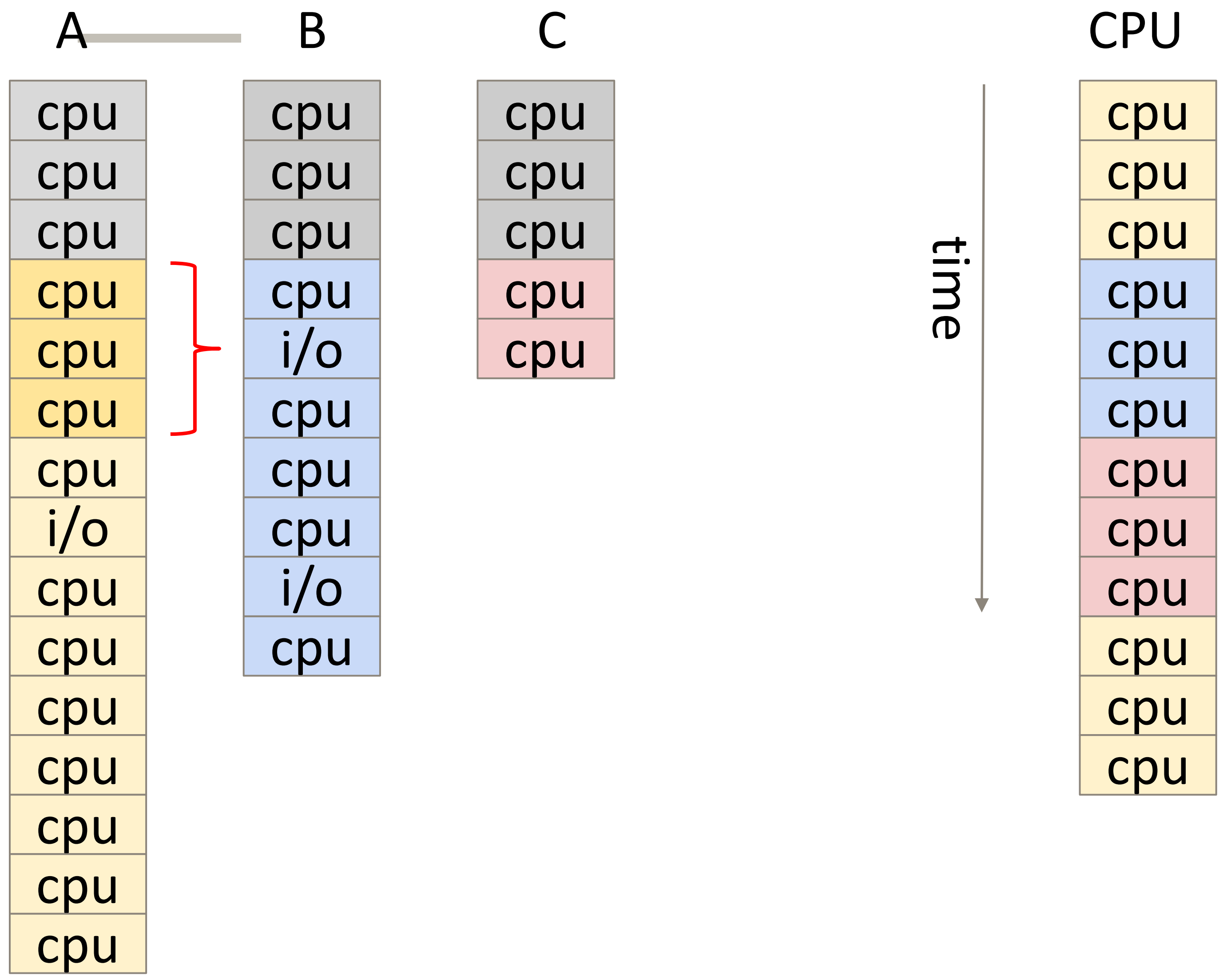
Exercise


A
B
C
CPU
time
A: cpu, cpu, cpu, cpu, cpu, cpu, cpu, i/o, cpu, cpu, cpu, cpu, cpu, cpu, cpu
B: cpu, cpu, cpu, cpu, i/o, cpu, cpu, cpu, i/o, cpu
C: cpu, cpu, cpu, cpu, cpu
CPU: cpu, cpu, cpu, cpu, cpu, cpu

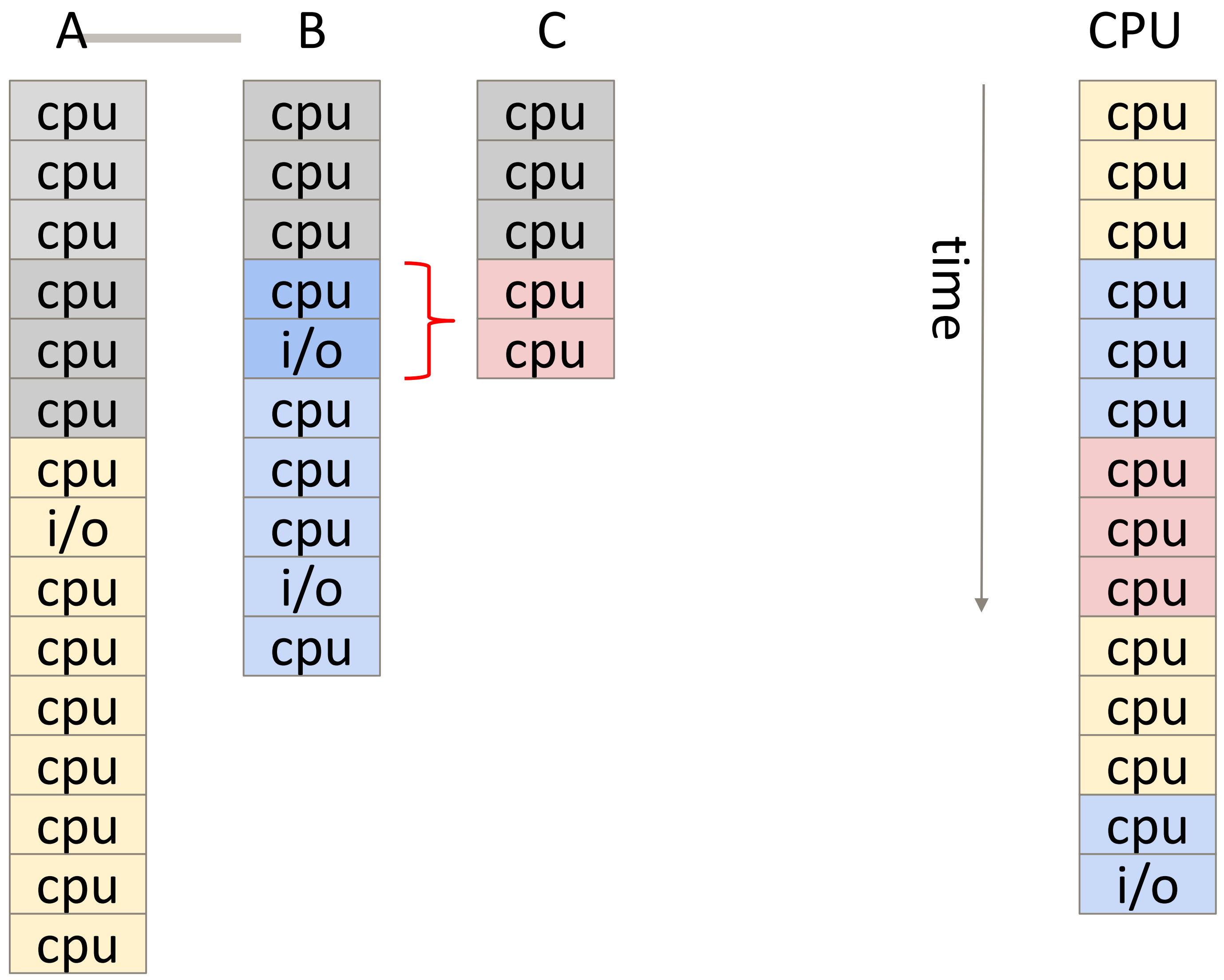
Exercise



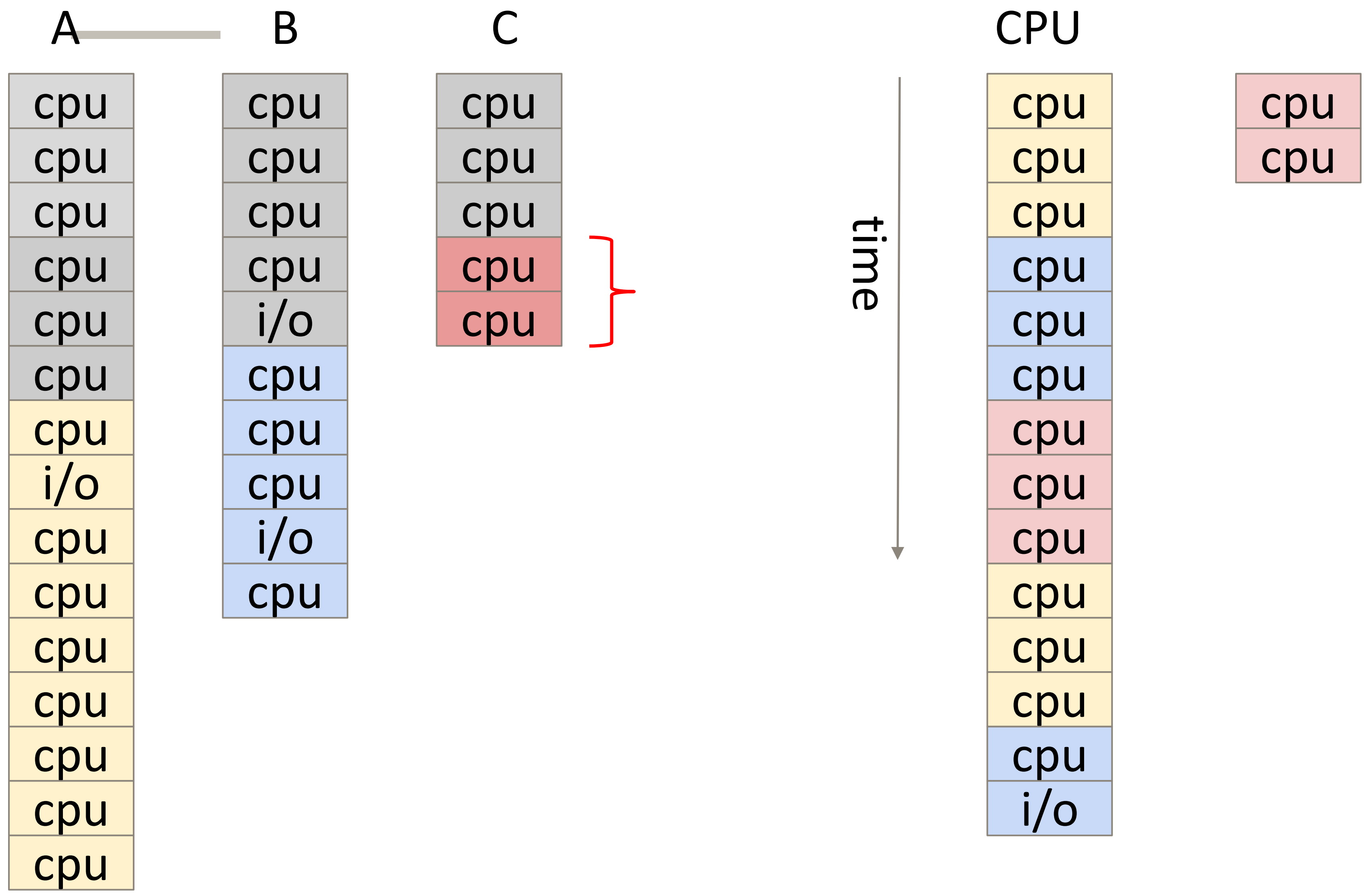
Exercise


A
B
C
CPU
time
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu

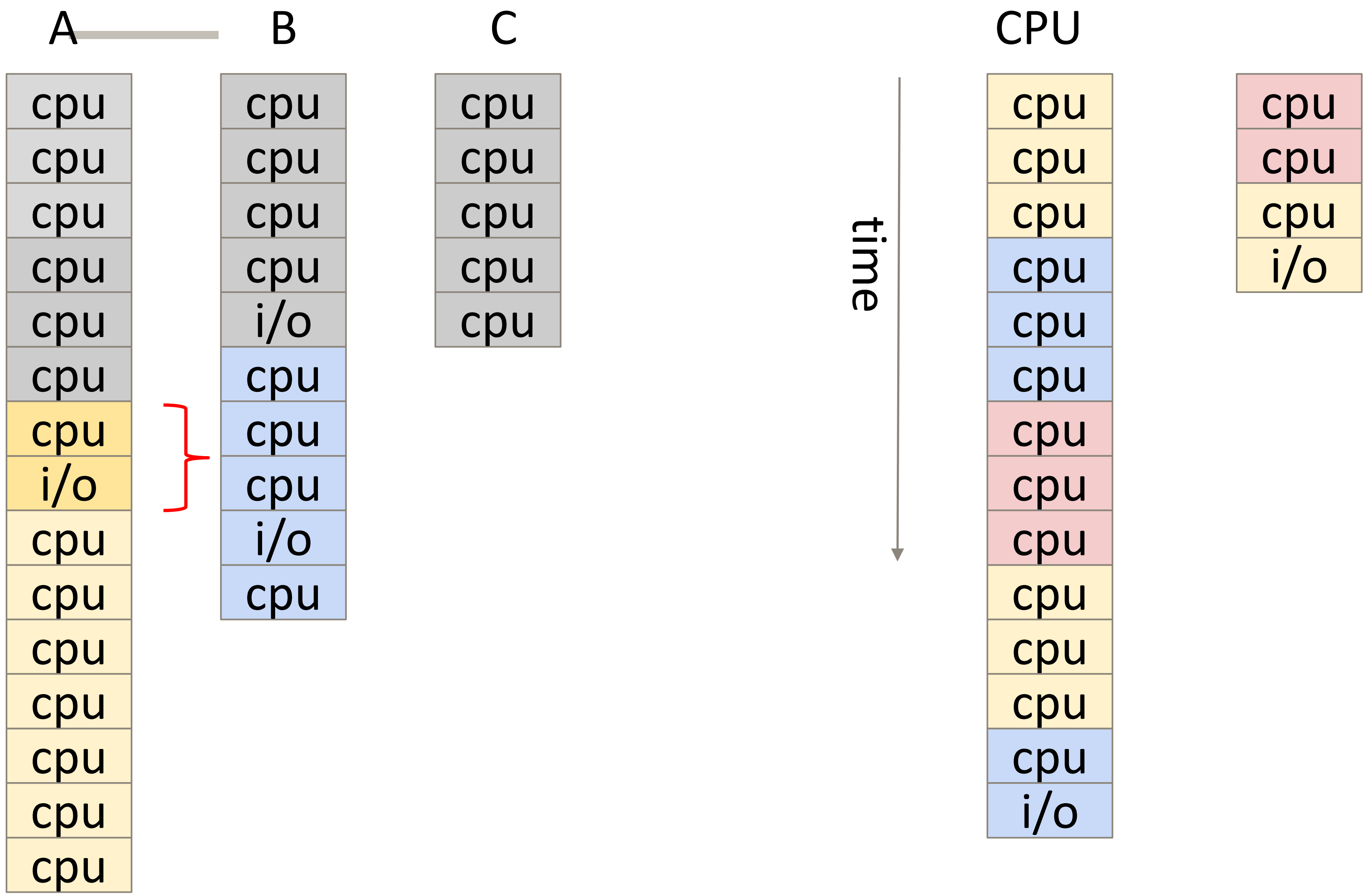
Exercise



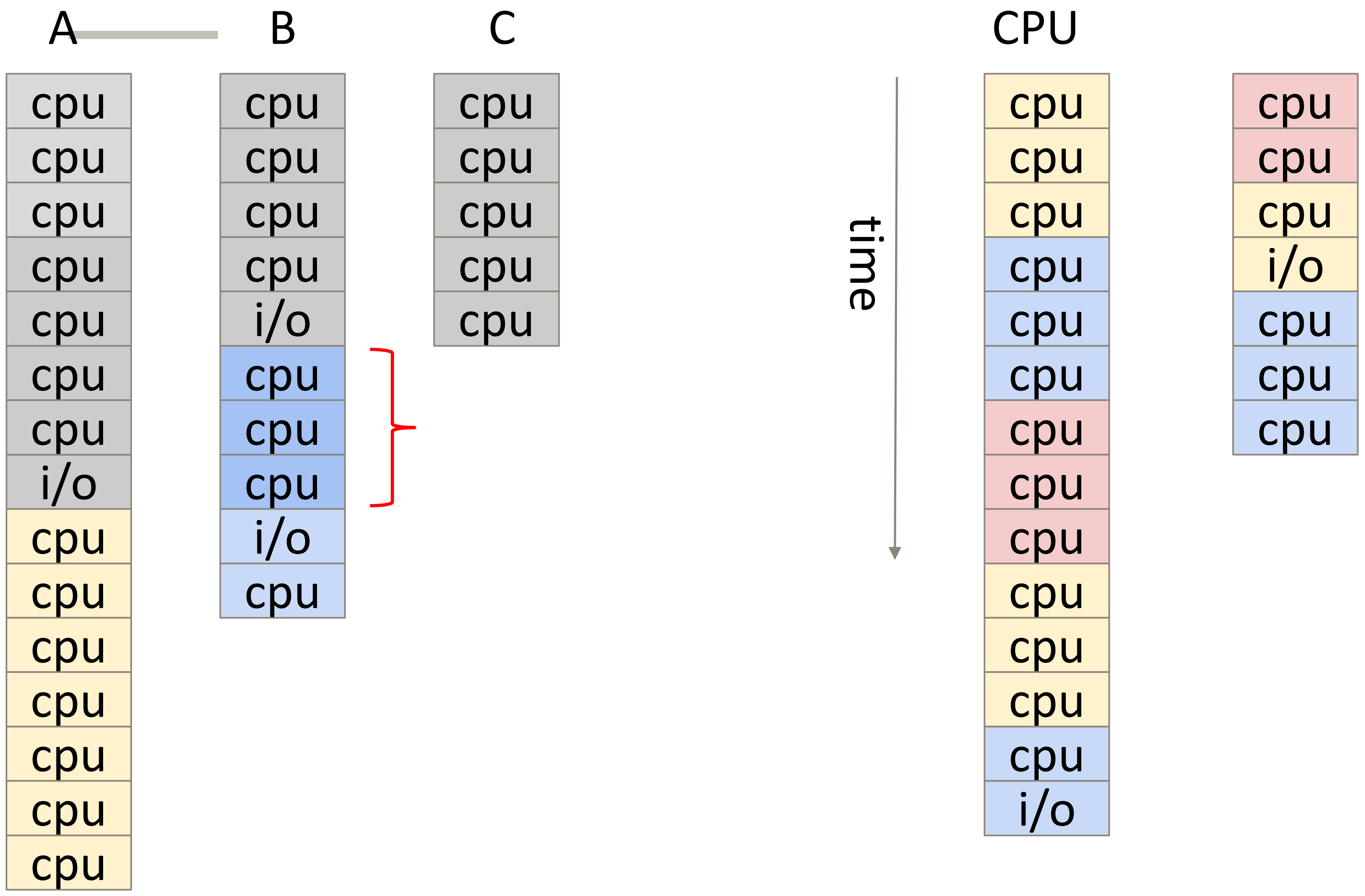
Exercise



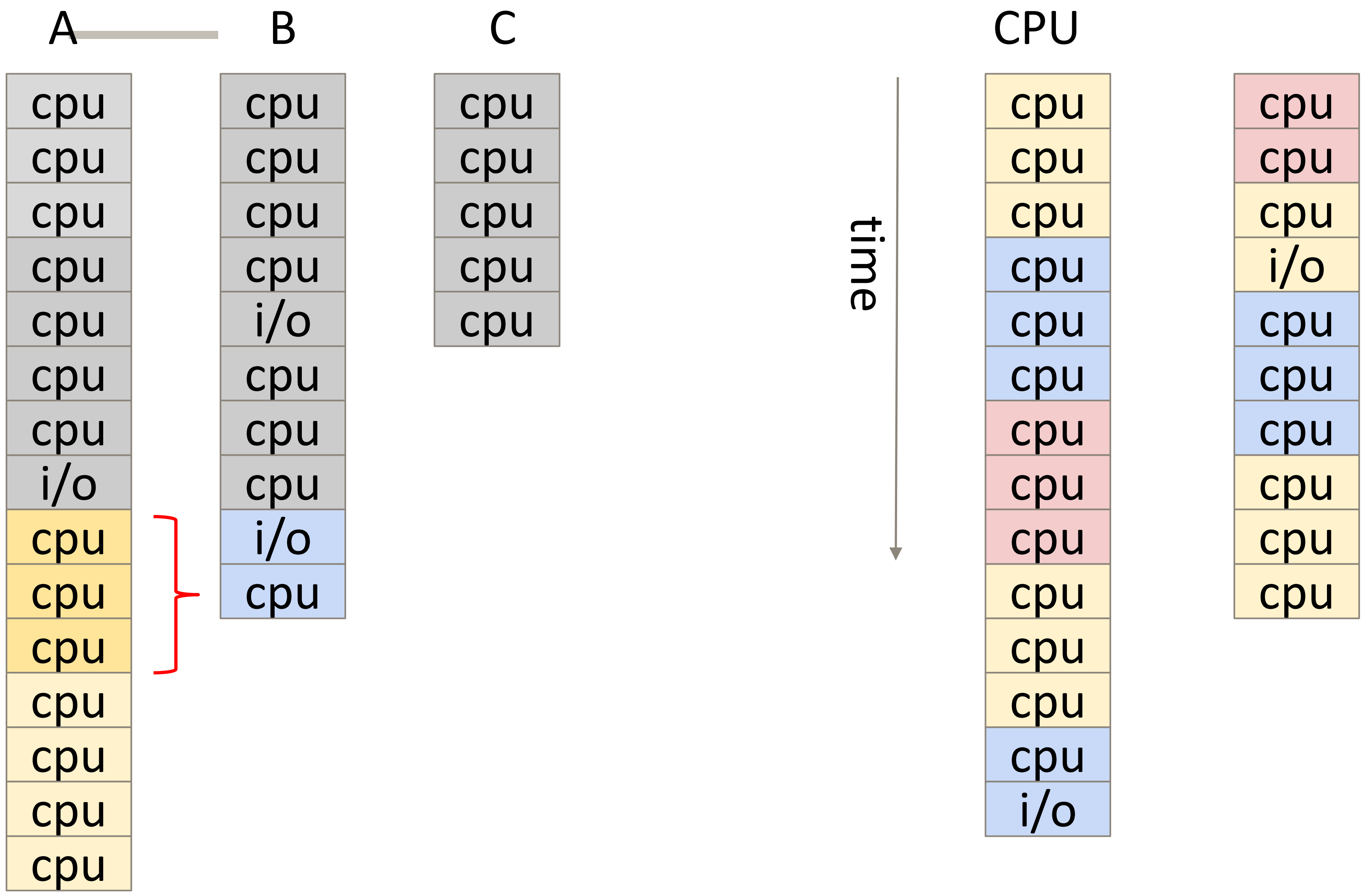
Exercise



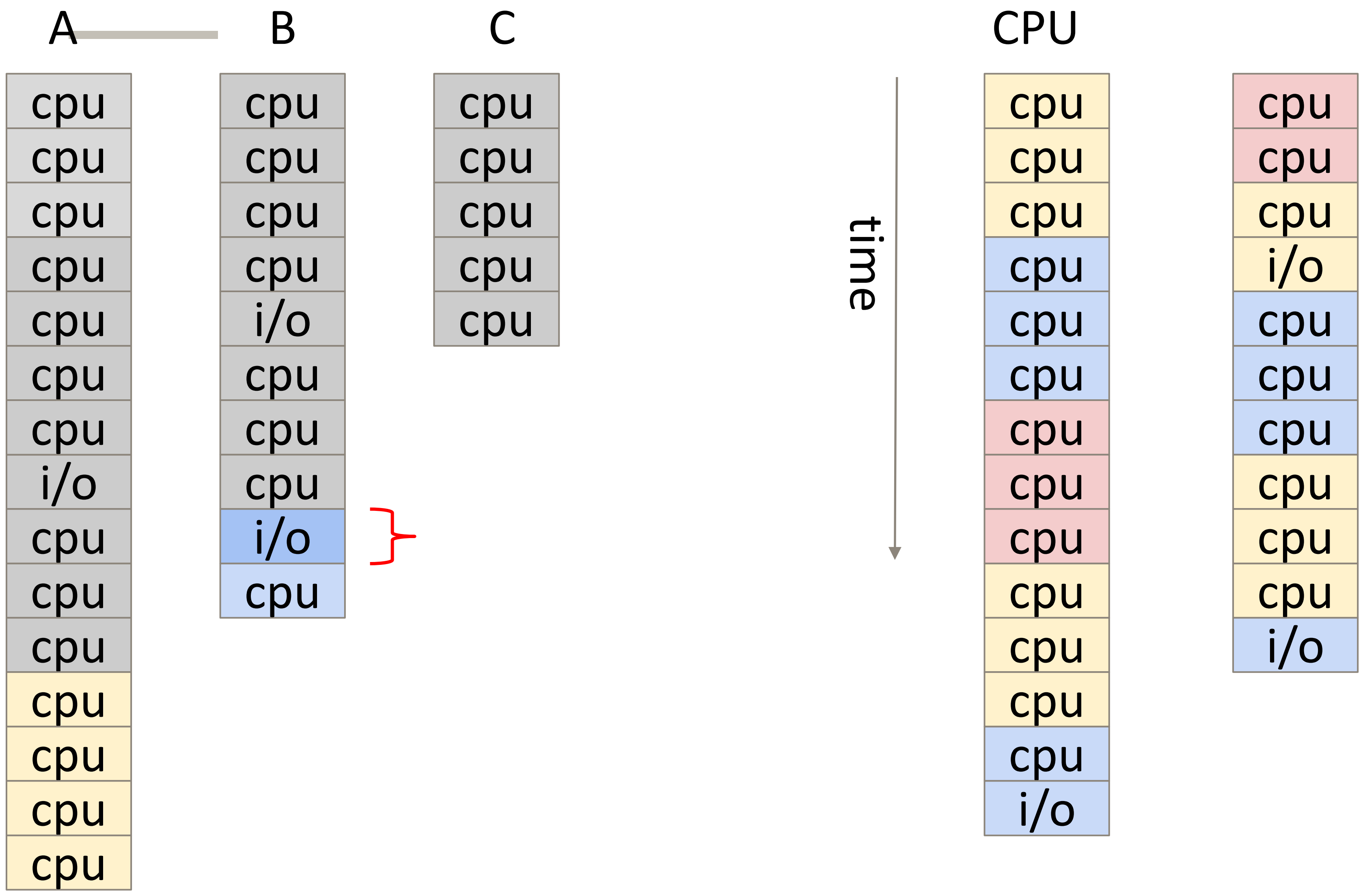
Exercise



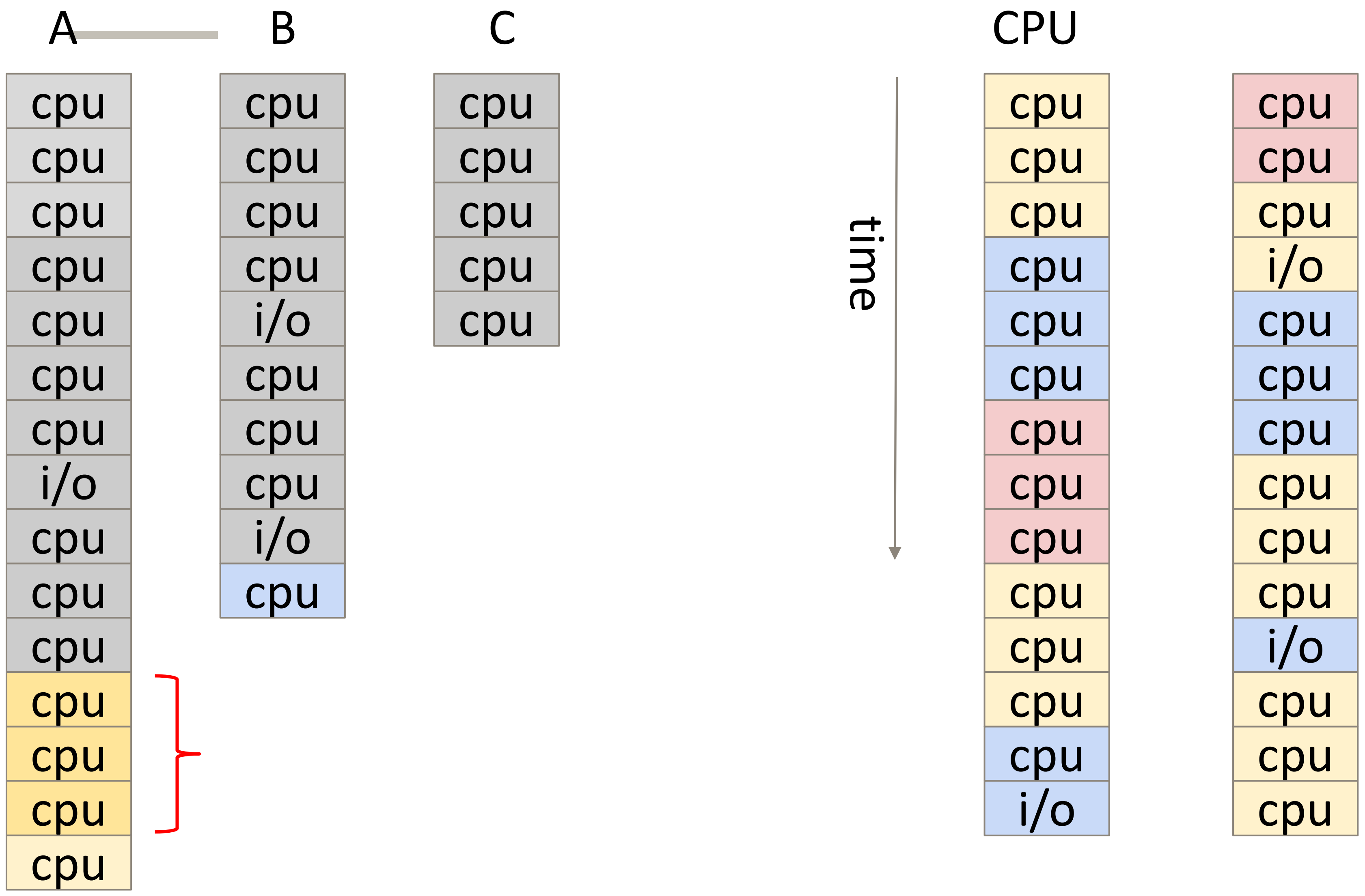
Exercise



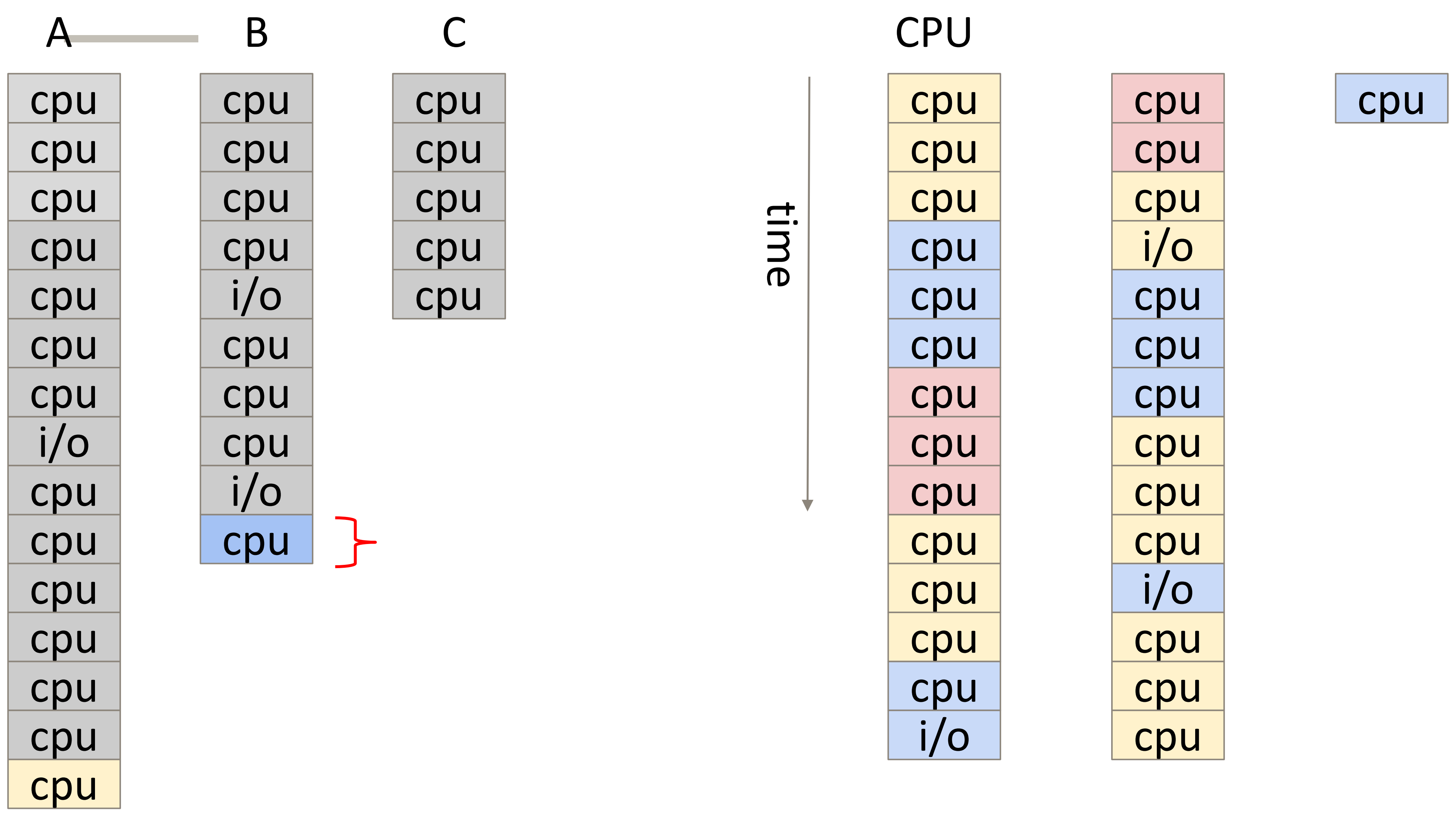
Exercise


A
cpu cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu cpu
B
cpu cpu cpu cpu i/o cpu cpu cpu i/o cpu
C
cpu cpu cpu cpu cpu
time
CPU
cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu i/o
cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o

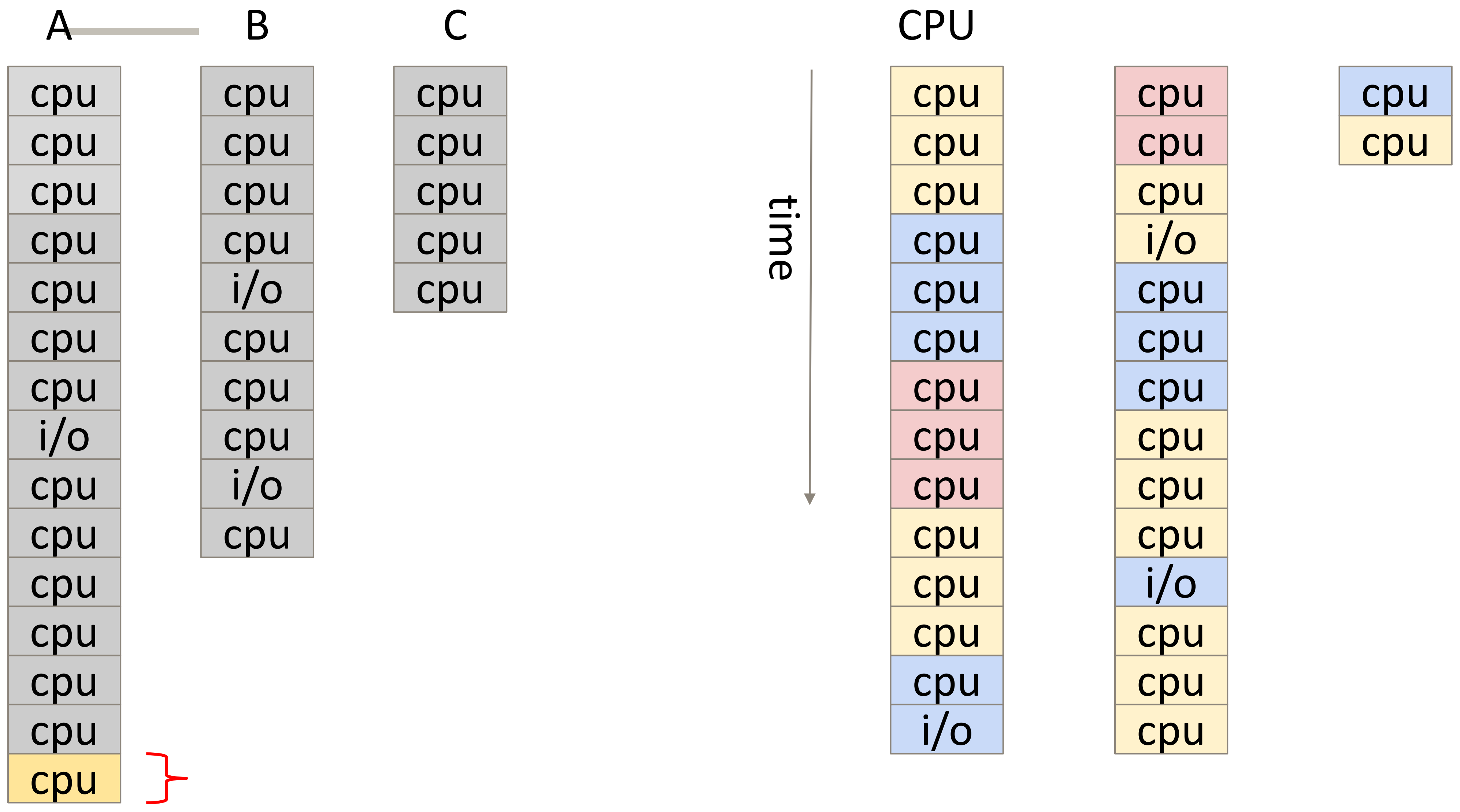
Exercise


A
B
C
CPU
time
A: cpu cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu cpu
B: cpu cpu cpu cpu i/o cpu cpu cpu i/o cpu
C: cpu cpu cpu cpu cpu
CPU: cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu i/o
CPU: cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu

Exercise


A
B
C
CPU
time
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu

Exercise


A
B
C
CPU
time
cpu cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu cpu
cpu cpu cpu cpu i/o cpu cpu cpu i/o cpu
cpu cpu cpu cpu cpu
cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu i/o
cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu
cpu cpu

Context Switching

UNIVERSITY OF CALGARY

Context switch

- essential part of any multitasking OS
- implements "OS takes CPU from one process, and gives it to another process“

- OS maintains a context (state) for each process
  - usually part of PCB, includes saved registers, open files, ...

- when OS switches between executing processes A to process B:
  - OS first saves A's state in A's PCB
    - e.g. save current CPU registers into PCB_A
  - OS then restores B's state from B's PCB
    - e.g. load CPU registers from PCB_B

- on multitasking systems, context switch is called many times every second, allowing the (illusion of) running more processes than the number of CPUs

Context switch

- context switch occurs in kernel mode:
  - for example when process exceeds its **time slice**
    - enforcing time slice policy usually implemented via timer interrupt
  - or when current process voluntarily relinquishes (**yields**) CPU, eg. by sleeping
  - or when current process requests a blocking I/O operation, or any blocking system call
  - or due to other events, such as keyboard, mouse, network interrupts

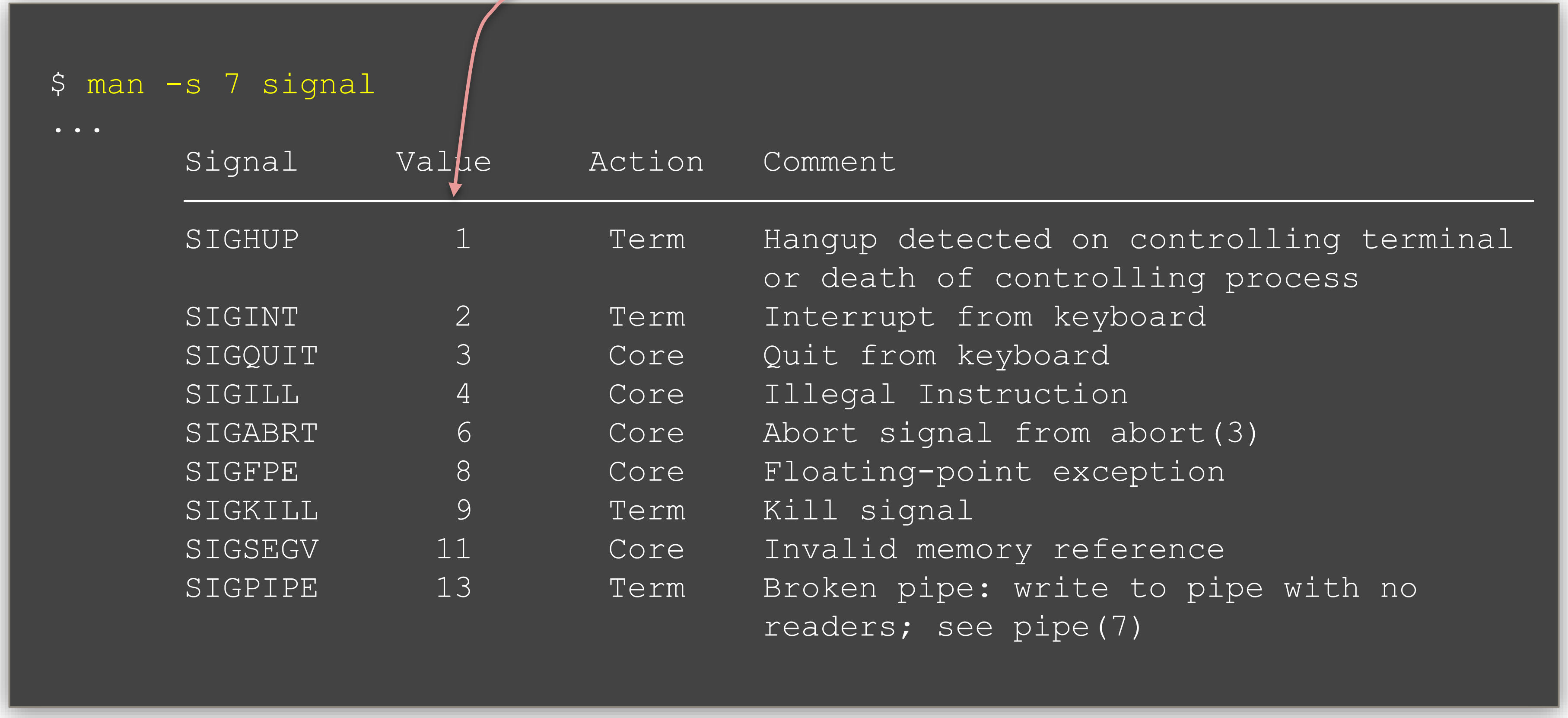
- context switch introduces time overhead
  - CPU spends cycles on no "useful" work, eg. saving/restoring CPU registers
  - context switch routine is one of the most optimized parts of kernels

- context switch performance can be improved with hardware support:
  - e.g. some CPUs support saving/restoring multiple registers in a single instruction,
    or CPU could support multiple sets of registers
  - software based context switch is slower, but more customizable, and often more efficient

Unix Signals

Unix signals

- a form of **interprocess communication** **(IPC)**
- similar concept to hardware interrupts on CPUs (you can think of it as process interrupt)
- very limited form of IPC - processes can send each other primitive messages
- the message is a single number, from a set of predefined integers



```
$ man -s 7 signal
...
       Signal       Value      Action    Comment
       ──────────────────────────────────────────────────────────────────────
       SIGHUP          1        Term     Hangup detected on controlling terminal
                                         or death of controlling process
       SIGINT          2        Term     Interrupt from keyboard
       SIGQUIT         3        Core     Quit from keyboard
       SIGILL          4        Core     Illegal Instruction
       SIGABRT         6        Core     Abort signal from abort(3)
       SIGFPE          8        Core     Floating-point exception
       SIGKILL         9        Term     Kill signal
       SIGSEGV        11        Core     Invalid memory reference
       SIGPIPE        13        Term     Broken pipe: write to pipe with no
                                         readers; see pipe(7)
```

Unix signals

- signals are used to notify a process that a particular event has occurred
  - one process (or thread) sends a signal, another process (or thread) receives it
  - it is possible for a process to signal itself
  - kernel can send signals to any processes

- signal lifetime:
  - a signal is **generated/sent**, usually as a consequence of some event
  - signal is **delivered/pending** to a process
  - delivered signal is **handled** by the process by executing a **signal handler**
  - some signals can be **ignored** - signal delivered to a process that ignores it is lost
  - some signals can be **blocked** - signal stays pending until it is unblocked

Generating signals

- manually from one process to another process
  `kill( pid, signal); // pid can be the current process`
- periodically via timer
  `alarm()` or `setitimer()`
- by kernel — to handle exceptions
  e.g. on segmentation fault kernel sends `SIGSEGV`

```
int main() {
  * (char *) 0 = 1;
}
```

- from command line

```
$ kill 12345    # tries to kill process w/pid=12345 by sending it signal
SIGTERM(15)
$ kill -9 12345 # kills a specific process with signal SIGKILL(9)
$ kill -9 -1    # kills all processes except pid=1 (init/systemd)
```

- more information on signals

```
      $ man -s 7 signal
```

Signal handling

- **signal handler** - a function that will be invoked when a signal is delivered
- **default signal handler** - all programs start with default handlers with default behaviours
- **a user-defined signal handler** - programs can override the default handlers
  - signals handled by a user-defined handler are called '**caught signals**'
- some signals cannot be caught – you cannot override their default signal handler
  - $ kill -9 pid always kills the process because
    SIGKILL(9) cannot be caught, blocked or ignored, and default handler kills the process
  - <ctrl-c> in a terminal will deliver SIGINT(2) to the running process, which can be caught, ignored or blocked
  - <ctrl-z> in a terminal will deliver SIGSTOP signal to the running process, which cannot be caught, ignored or blocked; default handler suspends the process

Signal handling

- signals can be delivered anytime, even while your program is in the middle of a function, or in the middle of applying an operator (C++)
- the state of your data might be in an inconsistent state
- signal handler could itself be interrupted !!!
- when writing signal handlers, keep it as simple as possible...
  - e.g. modify a global flag variable and let the program handle the interrupt later
  - declare global variables with volatile keyword, e.g.
    `volatile sig_atomic_t sigStatus = 0;`
  - only call reentrant functions inside the handler
  - more information and advanced tips, such as preventing signals from interrupting signal handlers:
    https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html

Re-entrant Functions

Re-entrant functions

- a function is re-entrant if:
    - □ it can be interrupted while in the middle of executing
    - □ and then called again (re-entered) from somewhere else
    - □ and finally the original function call can be resumed, and finish executing
- used in interrupt handlers, signal handlers, multi-threaded applications *
- when writing re-entrant functions:
    - be very careful with global variables and data structures
        - e.g. use atomic operations
    - do not call non-reentrant functions
        - unless you can temporarily disable interrupts / signals

Re-entrant functions

- example of a non-reentrant function:

```
int t;

void bad_swap(int *x, int *y)
{
    t = *x; // using a global variable t !!!
    *x = *y;
    // hardware interrupt or signal might
    // result in invoking (re-entering) swap()
here
    *y = t;
}
```

- easy to fix… can you guess how?

Re-entrant functions

- example of a re-entrant function:

```
void swap(int *x, int *y)
{
    int t = *x; // using a local variable t
    *x = *y;
    // hardware interrupt / signal here would be safe to call
swap() again
    *y = t;
}
```

- by using a local variable, the swap( ) function can be interrupted and re-entered anywhere
- please note that re-entrant functions, like the one above, are often also thread-safe, but not always
  see https://en.wikipedia.org/wiki/Reentrancy_(computing) for examples

Signal Handling Example

Signal handling example

```
#include <stdio.h> <stdlib.h> <signal.h> <unistd.h>

void sigint_handler( int signum )
{
  printf("\ncaught signal=%d\n", signum);
  printf("LOL, you think <ctrl-c> will stop me?!!?!\n");
}

int main (int argc, char *argv[])
{
  /* catch <ctrl-c> and laugh at the user */
  signal(SIGINT, sigint_handler);

  for(int i = 1 ; i < 10 ; i++) {
    printf("Loop=%d\n", i);
    sleep( 1 );
  }
  printf("Exiting now.\n");
  exit(0);
}
```

Output:

```
$ ./a.out
Loop=1
Loop=2
Loop=3
Loop=4
Loop=5
Loop=6
Loop=7
Loop=8
Loop=9
Exiting now.
```

Signal handling example

```
#include <stdio.h> <stdlib.h> <signal.h>

void sigint_handler( int signum )
{
  printf("\ncaught signal=%d\n", signum);
  printf("LOL, you think <ctrl-c> will st
}

int main (int argc, char *argv[])
{
  /* catch <ctrl-c> and laugh at the user
  signal(SIGINT, sigint_handler);

  for(int i = 1 ; i < 10 ; i++) {
    printf("Loop=%d\n", i);
    sleep( 1 );
  }
  printf("Exiting now.\n");
  exit(0);
}
```

Possible output:

```
$ ./a.out
Loop=1
Loop=2
Loop=3
^C
caught signal=2
LOL, you think <ctrl-c> will stop
me?!!?!
Loop=4
Loop=5
Loop=6
^C
caught signal=2
LOL, you think <ctrl-c> will stop
me?!!?!
Loop=7
Loop=8
Loop=9
Exiting now.
```

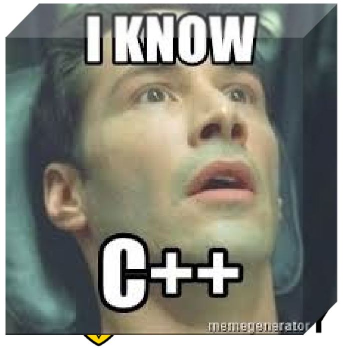
Swap

swap( ) in C and C++

```c
/* swap in C
 * pointers are ugly
 */

void
swap(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}

int a, b;
swap( &a, &b);
```

```cpp
// swap in C++
// references are
cool


void
swap(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}

int a, b;
swap( a, b);
```

```cpp
// swap in C++
// templates are cool


template <class T>
void swap(T &x, T &y)
{
    T t(x);
    x = y;
    y = t;
}

double a, b;
swap( a, b);
std::vector<int> c, d;
swap( c, d);
```



CPU Utilization Example

CPU utilization

- example:
  - OS is running 4 processes, P1, P2, P3 and P4
    - P1 spends **40%** of the time waiting on I/O
    - P2 spends **20%** of the time waiting on I/O
    - P3 spends **50%** of the time waiting on I/O
    - P4 spends **90%** of the time waiting on I/O
  - if there is only one CPU, what will be the CPU utilization?

    i.e. what percentage of the time is the CPU going to be running 'something'?
- Answer:
  - CPU utilization = probability that at least one of the processes is not waiting on I/O

    = ? ? ?

CPU utilization

- example:
  - OS is running 4 processes, P1, P2, P3 and P4
    - P1 spends **40%** of the time waiting on I/O
    - P2 spends **20%** of the time waiting on I/O
    - P3 spends **50%** of the time waiting on I/O
    - P4 spends **90%** of the time waiting on I/O
  - if there is only one CPU, what will be the CPU utilization?

    i.e. what percentage of the time is the CPU going to be running 'something'?
- Answer:
  - CPU utilization = probability that at least one of the processes is not waiting on I/O

    = 1.0 - probability that all processes are waiting on I/O

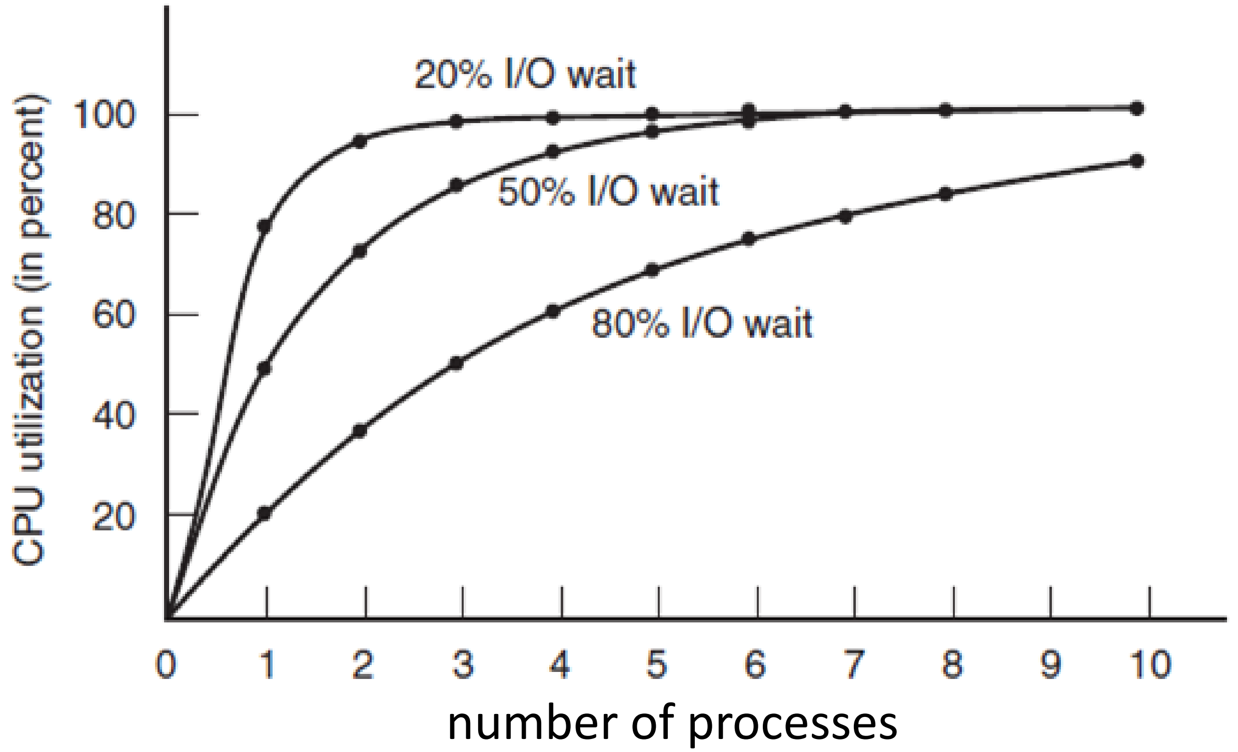
    = 1.0 - 0.4 * 0.2 * 0.5 * 0.9 = 0.964

    = 96.4%

CPU utilization - under simplistic multiprogramming model

CPU utilization as a function of the number of processes in memory.

- assume ***n*** similar processes
- each process spends the same fraction ***p*** of its time waiting on I/O
- then CPU utilization = ***1 - pⁿ***



CPU utilization example

- example:
  - computer has 8GB of RAM
  - 2GB are taken up by OS, leaving 6GB available to user programs
  - user wants to run multiple copies of a program that needs 2GB RAM, with average 80% I/O
  - with 6GB remaining, user could run 3 copies of the program
  - CPU utilization would be = $1 - 0.8^3$ ~= 49%

- is it a good idea to buy 8GB more of RAM?

CPU utilization example

- example:
  - computer has 8GB of RAM
  - 2GB are taken up by OS, leaving 6GB available to user programs
  - user wants to run multiple copies of a program that needs 2GB RAM, with average 80% I/O
  - with 6GB remaining, user could run 3 copies of the program
  - CPU utilization would be = $1 - 0.8^3$ ~= 49%

- is it a good idea to buy 8GB more of RAM?
  - with 14 GB available, we could run 7 copies of the program
  - CPU utilization would be = $1 - 0.8^7$ ~= 79%
  - throughput increased by 79% - 49% = **30%**

- is it a good idea to buy 8 GB more?
  - we could run 11 programs ⟶ CPU utilization = $1 - 0.8^{11}$ ~= 91%
  - throughput increased only by 91% - 79% = **12%** (diminishing returns)

UNIVERSITY OF CALGARY

Process Creation

Process creation

- in UNIX
  - `init` process is created at boot time by kernel (special case)
    - in most modern Linux distributions `init` was replaced by `systemd`
  - afterwards, only an existing process can create a new processes, via `fork()`
  - therefore all other processes are descendants of `init`

- in Windows:
  - `CreateProcess()` is used to create processes
  - but the behavior is quite different from `fork()`

Process creation

- typical scenarios when process need to create new process[es]:
  - during system initialization (boot)
    - spawning background processes — daemons, services, e.g. database server
    - calling custom scripts
  - application decides to spawn additional processes
    - e.g. to execute external programs or to do parallel work
  - a user requests to create a new process
    - e.g. window manager allows users to launch applications
  - starting batch jobs
    - mainframes

Signals

Recommendations:

- avoid signals as an IPC mechanism if you can
- especially in multi-threaded programs
- use signals only if you 'have to', eg. for background processes
- more info on signals & C++

  https://en.cppreference.com/w/cpp/utility/program/signal

Resource allocation

- several options for allocating resources for a new process, for example:
  - child obtains resources directly from the OS
    - most common, easiest to implement
    - every new process gets the same resources
    - fork bomb crashes the system
  - child obtains subset of parent's resources
    - parent must give some of its resources to child
    - fork bomb has limited impact
  - parent shares resources with the child – e.g. with threads
  - hybrids

Review

Review

- Which one of the following executes in kernel mode?
  - A user program
  - A library function call
  - A system call
  - A system call wrapper function

- In C, printf() is a system call.
  - True
  - False

Review

- When 4 programs are executing on a computer with a single CPU, how many program counters are there?
- When does a program become a process?

Review

- What is the name of the PCB data structure in Linux?
- Name some of fields in a PCB.
- On UNIX systems, what is the name of the process that is the ancestor of all user processes?

Summary

- Multi-programming vs. Multi-tasking
- Program vs. Process
- Forking
- External Programs
- More PCB
- Process Management & Scheduling
- Context switching
- Unix Signals
- Re-entrant Functions
- Signal Handling
- CPU Utilization
- Process Creation

Onward to …
Basic File Systems

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY