CPSC 457

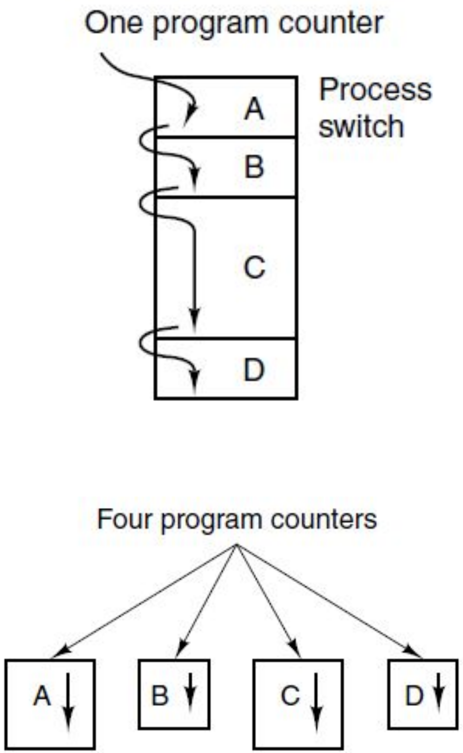
Processes - part 2

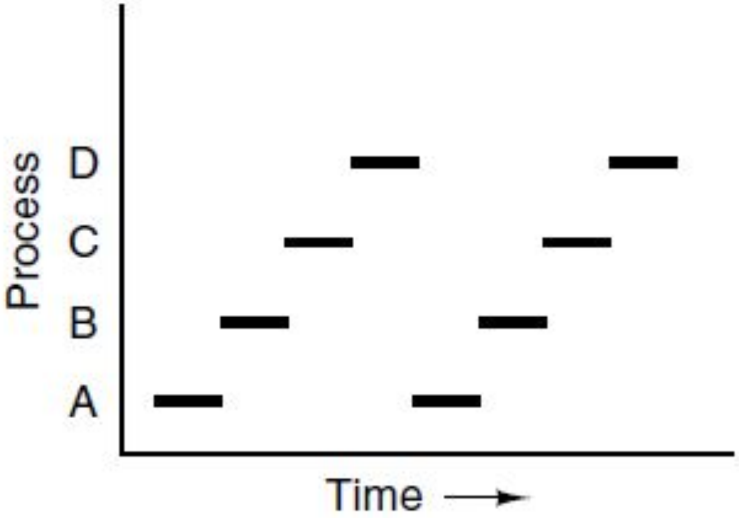
Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- CPU utilization
- processes creation, termination
- process scheduling
- process states
- context switching
- signals

Multiprogramming on a single CPU





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?

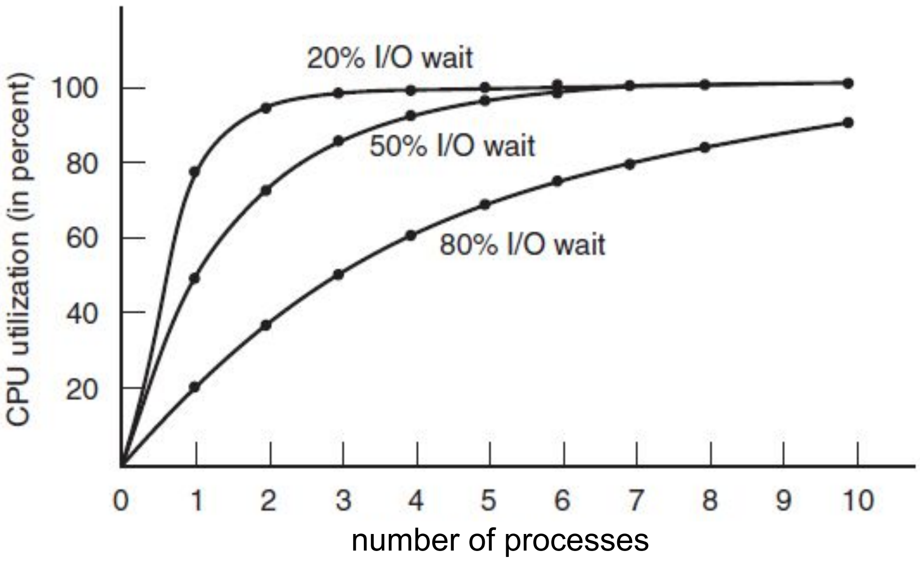
    i.e. what percentage of the time is the CPU going to be running 'something'?
- Answer:
  - CPU utilization = probability that at least one of the processes is not waiting on I/O

    = 1 - (probability that all processes are waiting on I/O)

    = 1 - (0.4 * 0.2 * 0.5 * 0.9) = 0.964 = 96.4%

CPU utilization - under simplistic multiprogramming model

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



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

Tanenbaum & Bo,Modern Operating Systems:4th ed.

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 = $1 - 0.8^3 \sim= 49\%$
- is it a good idea to buy 8GB more of RAM?
  - with 14GB available, we could run 7 copies of the program
  - CPU utilization would be = $1 - 0.8^7 \sim= 79\%$
  - throughput increased by 79% - 49% = **30%**
- is it a good idea to buy 8GB more?
  - we could run 11 programs $\longrightarrow$ CPU utilization = $1 - 0.8^{11} \sim= 91\%$
  - throughput increased only by 91% - 79% = **12%** (diminishing returns)

Process creation

- in UNIX
  - `init` process is created at boot time by kernel (special case)
  - afterwards, only an existing process can create a new processes, via `fork()`
  - therefore all other processes are descendants of `init`
  - in many modern Linux distributions `init` is replaced by `systemd`
  - `fork()` often followed by `exec*()` to spawn a different program
    (remember, you you can use `system()` and `popen()` convenience functions)

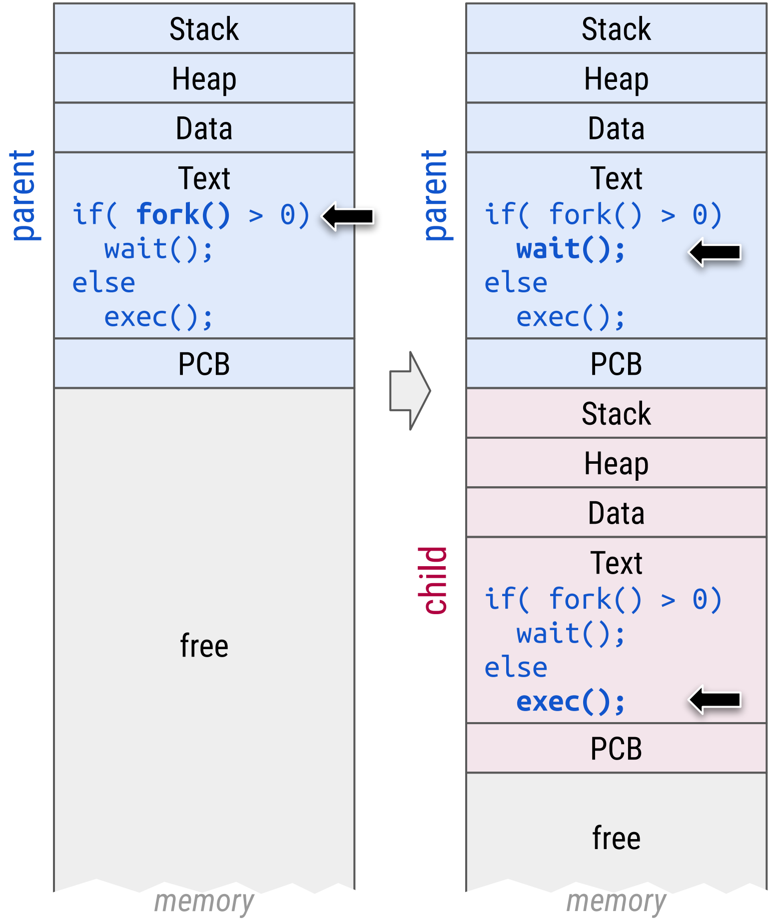
- in Windows: **`CreateProcess()`** is used to create processes,
  but the behavior is quite different from **`fork()`**

Process creation

- there a many reasons for a process to create new process[es]
- during system initialization (boot)
  - spawning background processes — daemons, services, eg. database server
- application decides to spawn additional processes
  - eg. to execute external programs or to do parallel work
- a user requests to create a new process
  - eg. GUI windows manager application allows launching new applications
- starting batch jobs
  - mainframes

Address space

- each process has its own address space
- `fork()` duplicates address space, creating nearly identical copy of itself
- next instruction is the same (`if`)
- but code flow may differ for child vs. parent



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 – eg. with threads
  - hybrids

Common parent-child execution scenarios

- after child is created, parent usually does one of 3 things:
  1. the parent waits until the child process is finished
     - often used when child executes another program, eg. **`fork/exec()`**, or **`system()`**
  2. the parent continues to execute concurrently and independently of the child process
     - eg. autosave feature
  3. the parent continues to execute concurrently, but synchronizes with the child
     - can be quite complicated to synchronize

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

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

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

Process termination

- typical reasons for terminating a process
- 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, eg. accessing invalid memory, division by zero
  - **external** – killed by another process
    - parent, or another process calls `kill()`
    - eg. during shutdown, pressing <ctrl-c> in terminal, closing GUI window

Process termination

- parent may 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
- in Unix, when a parent process is terminated:
  - 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 (eg. to kill children, reparent to some other process)
  - see `$ man prctl` for more details

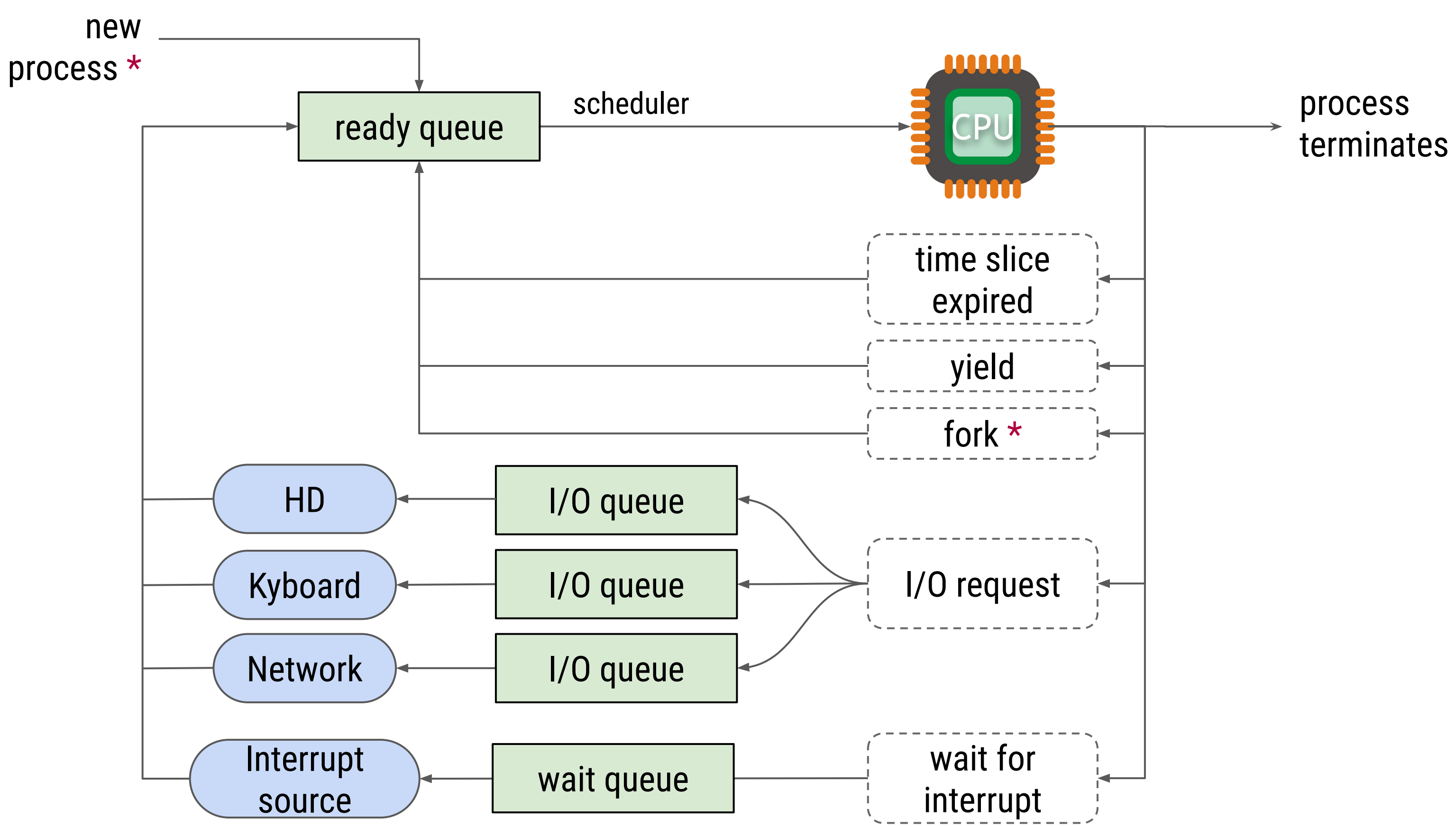
Process termination

- when terminating a process the OS mus 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

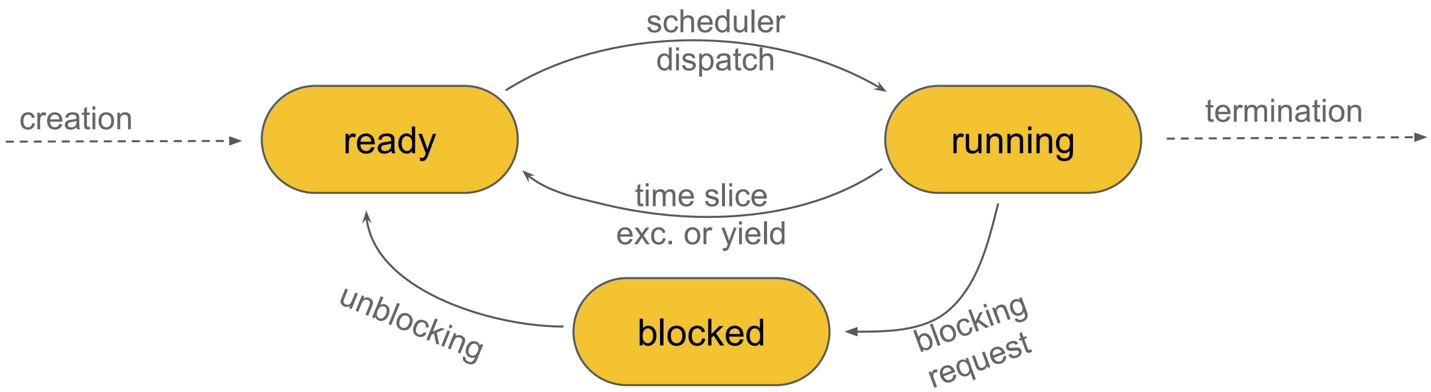
- part of multitasking is deciding which process gets the CPU next
- typical objective is to maximize CPU utilization
- **process scheduler**:
  - kernel routine/algorithm that chooses one of available process to execute next on the CPU
  - selected from processes in a ready queue
- OS maintains different scheduling queues:
  - job queue: all programs waiting to run, usually found in batch systems
    - eg. priority queue
  - ready queue: all processes that are ready to execute their next instruction
    - eg. linked list, priority queue, … depends on scheduler
  - device queues: processes waiting for a particular device
    - each device has its own queue

Process scheduling diagram



Process states

3 process states:	only 4 transitions are possible:
■ **running** — actually running on the CPU	■ ready → running
■ **blocked** — waiting for some event to occur, eg. I/O	■ running → ready
■ **ready** — the process is ready to execute on CPU	■ running → blocked
	■ blocked → ready

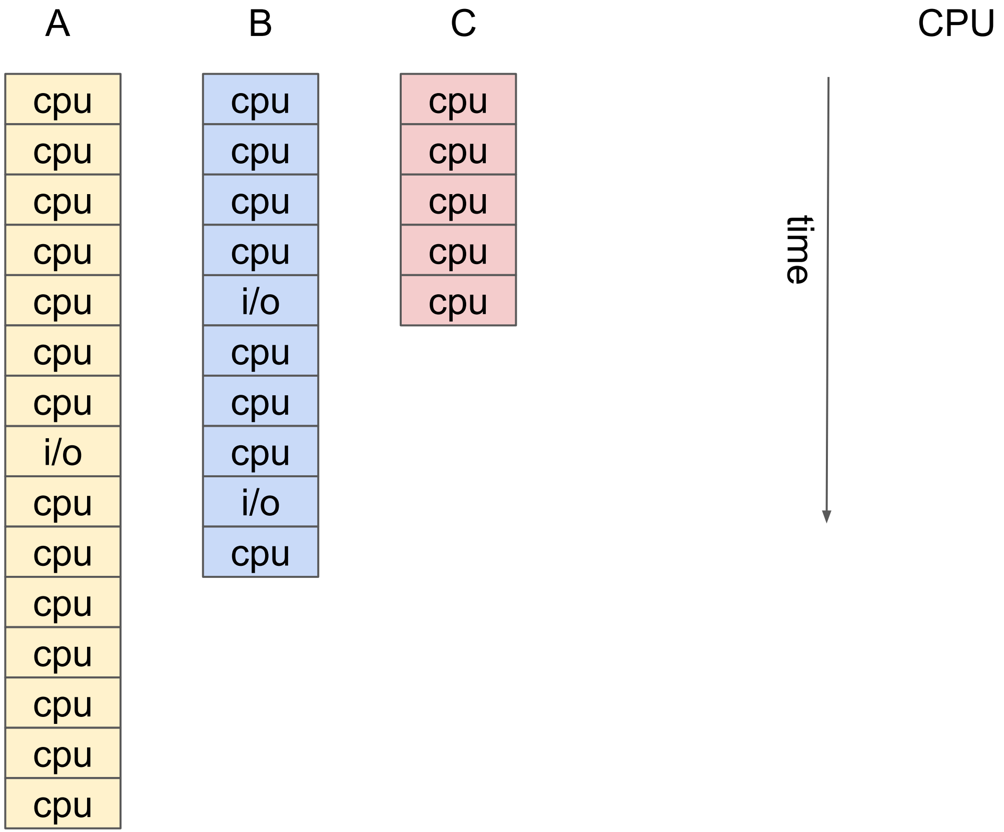


Exercise – simulating round-robin scheduling

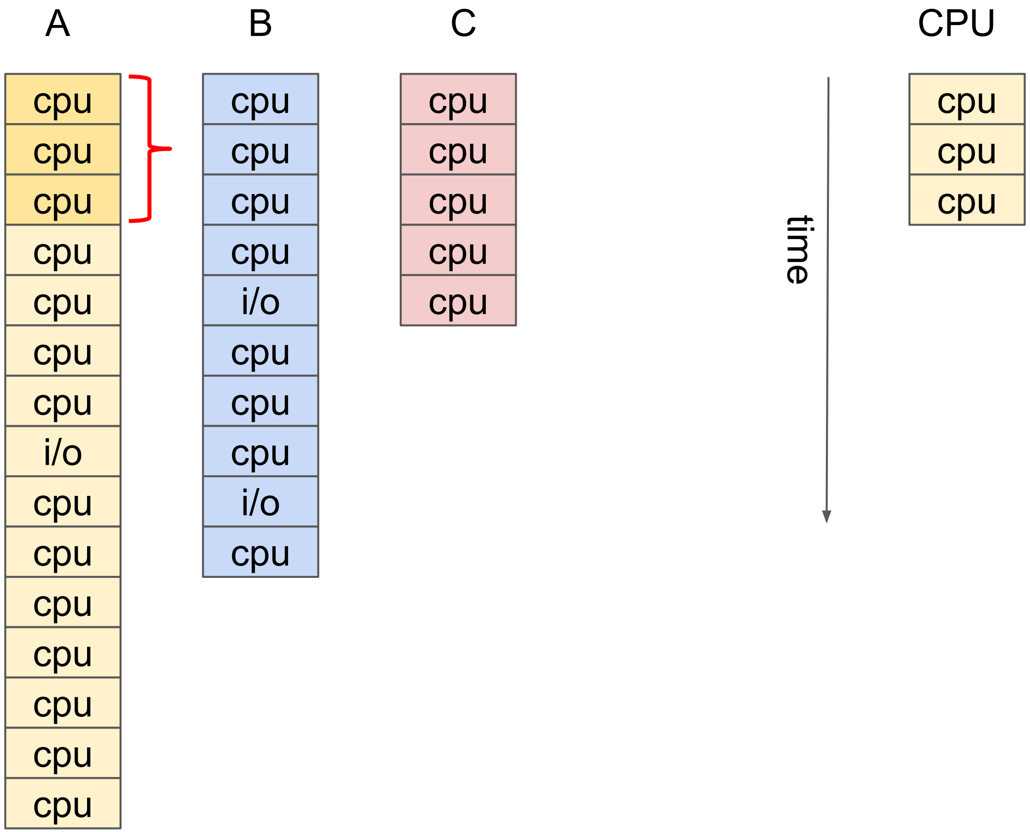
- simulate 3 processes A, B, C
  - A: 7 units of CPU, 1 unit of I/O, 7 units of CPU
  - B: 4 CPU, 1 I/O, 3 CPU, 1 I/O, 1 CPU
  - C: 5 CPU
- assume time slice of 3 units
  - each process gets 3 units of CPU cycles
  - if process requests I/O during its time slice, OS switches to the next process
  - otherwise,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

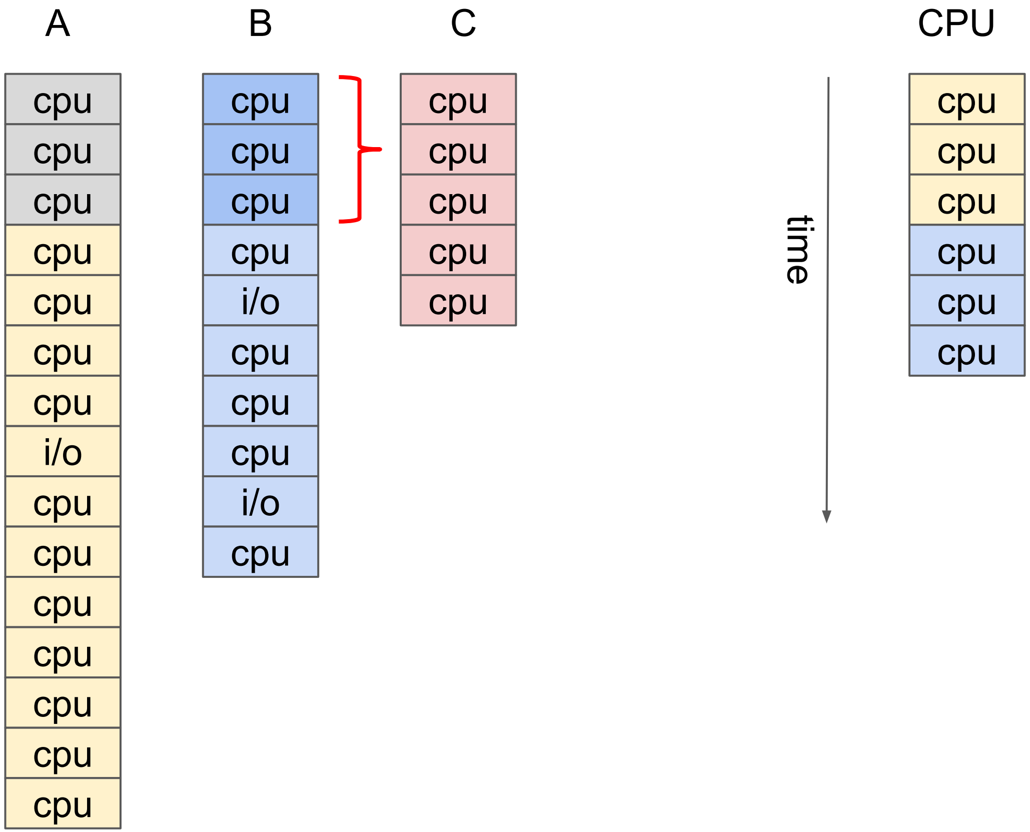
Exercise


A
B
C
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
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
time

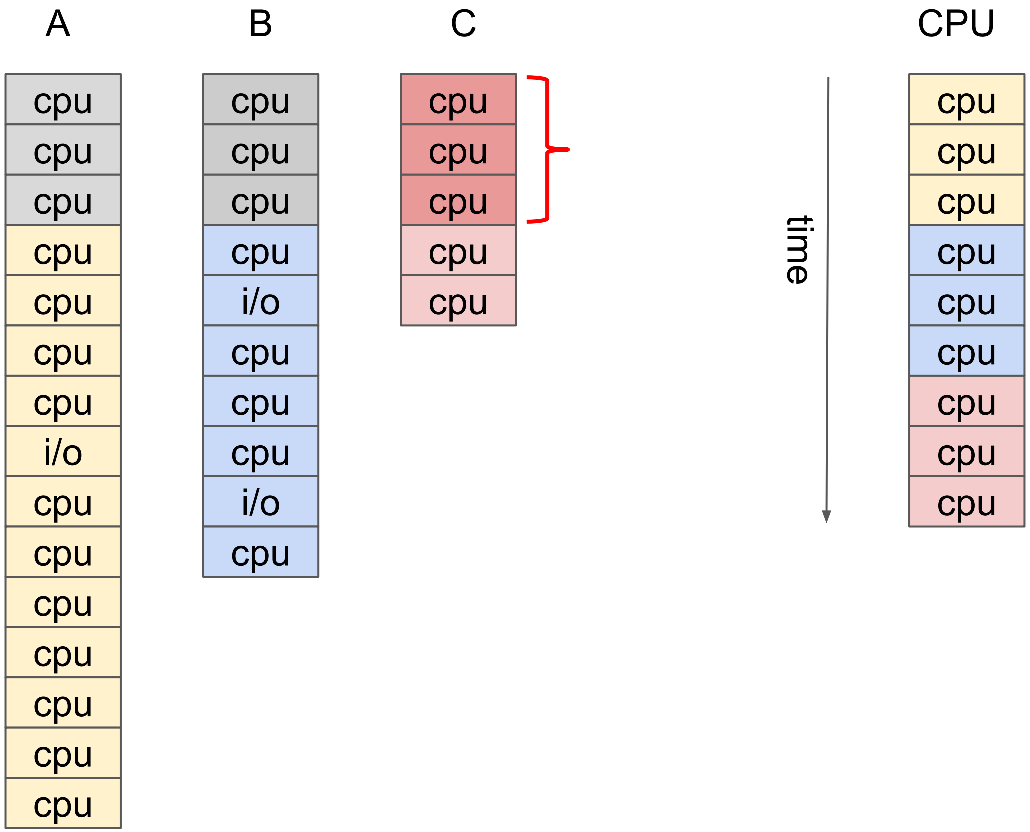
Exercise


A
B
C
CPU
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
time
cpu
cpu
cpu

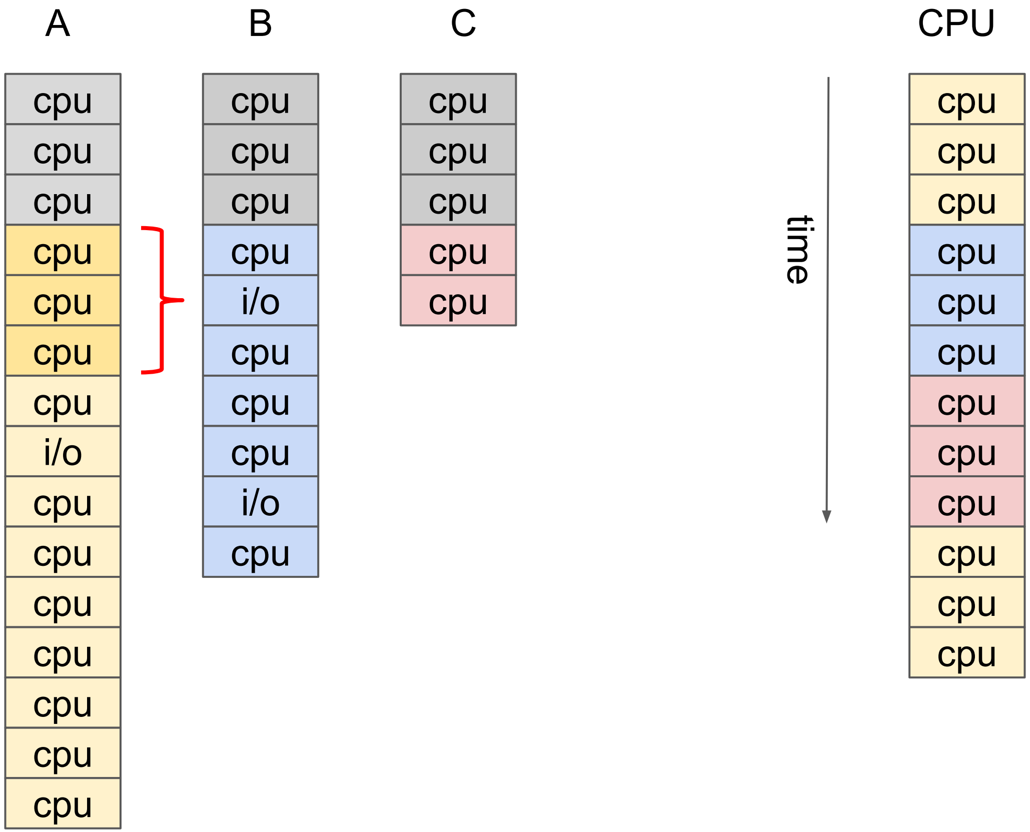
Exercise


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

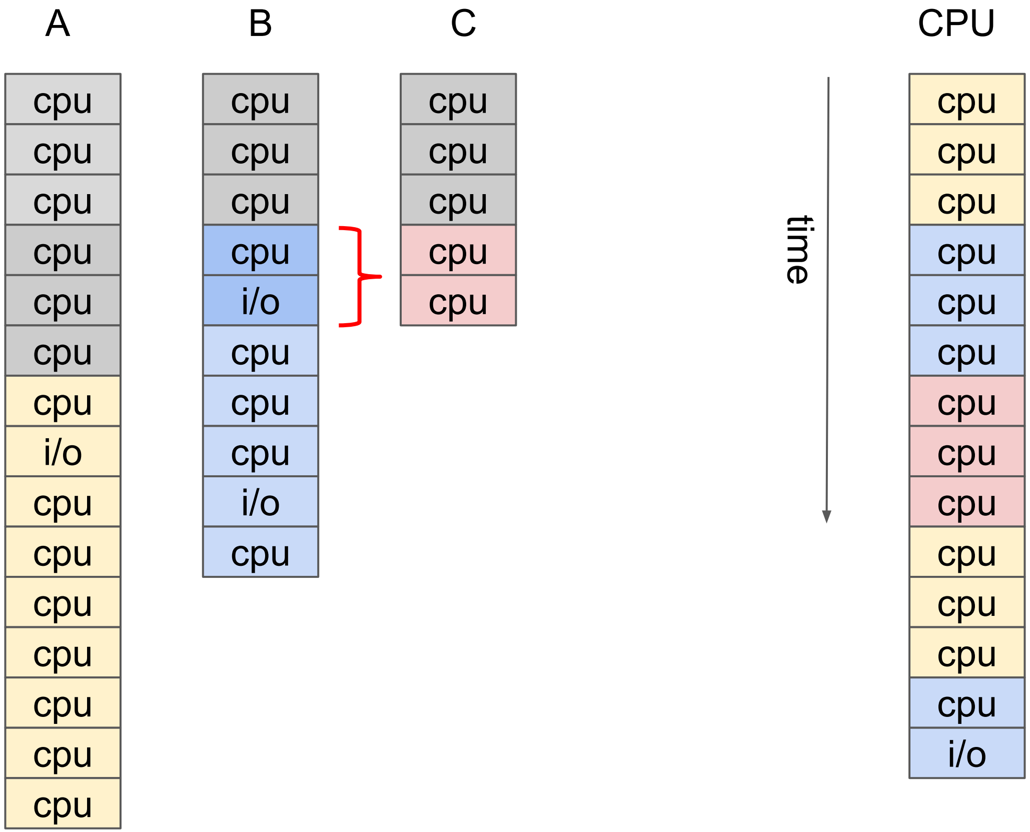
Exercise


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

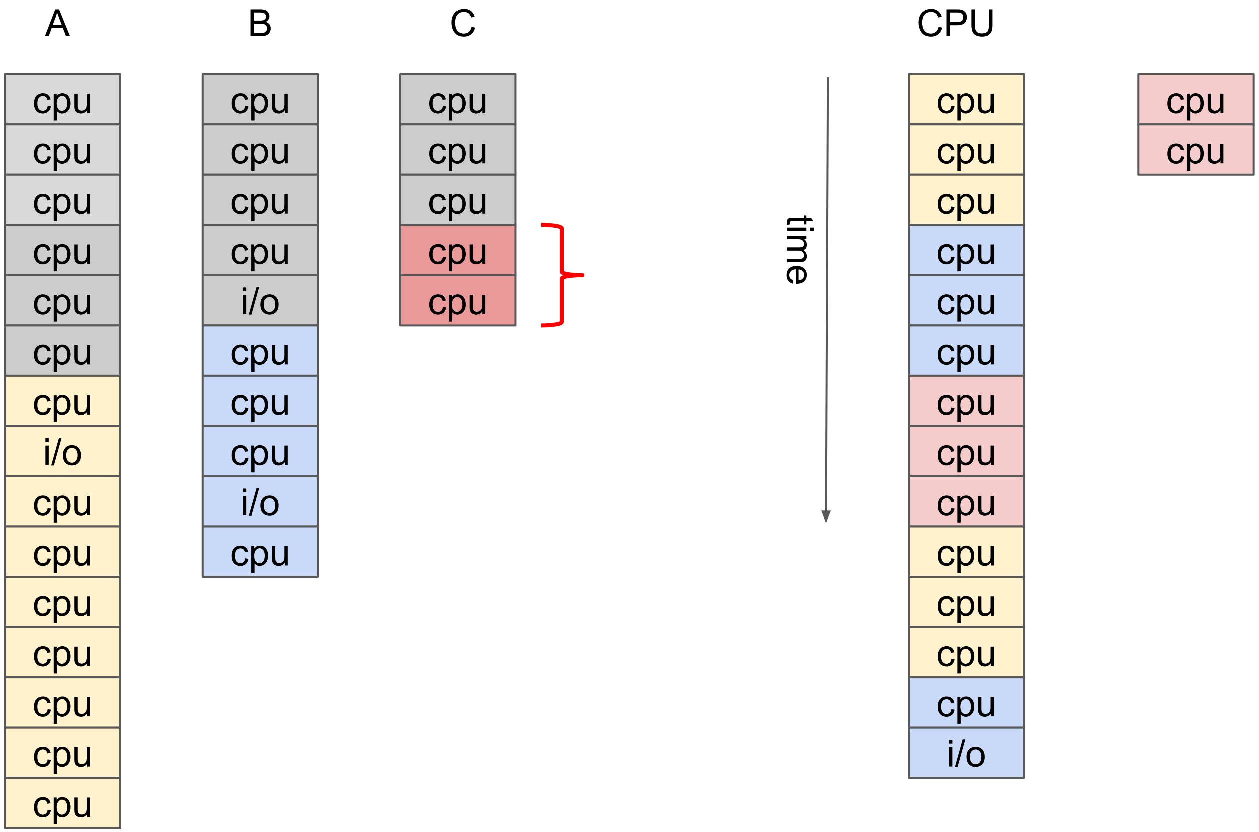
Exercise


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

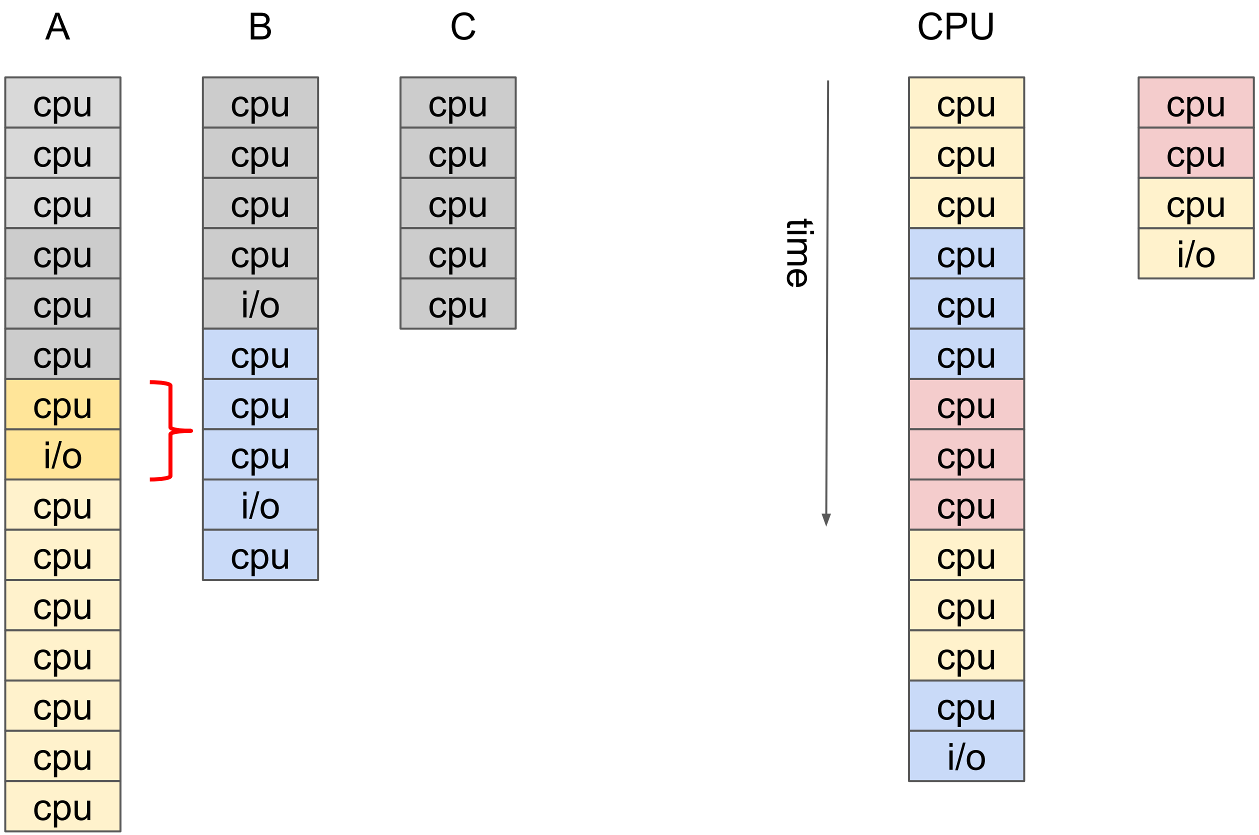
Exercise



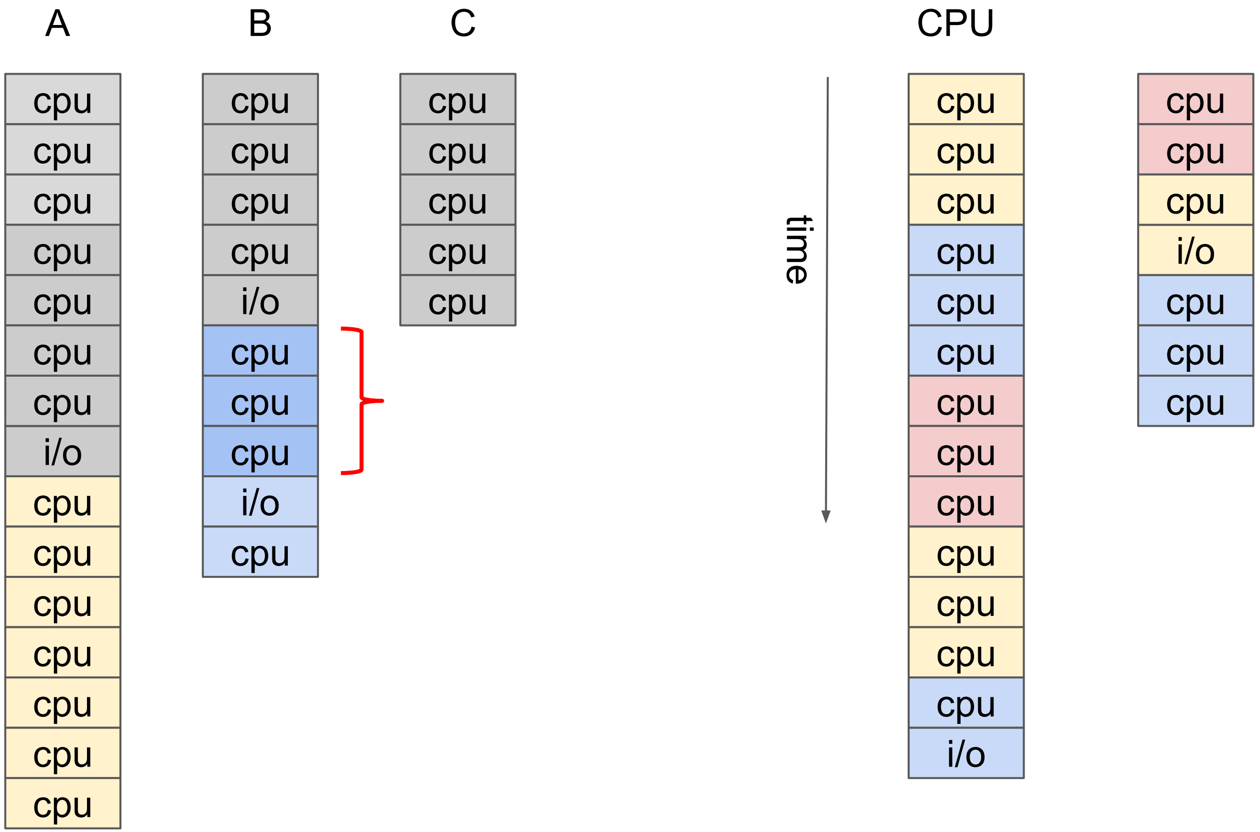
Exercise


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

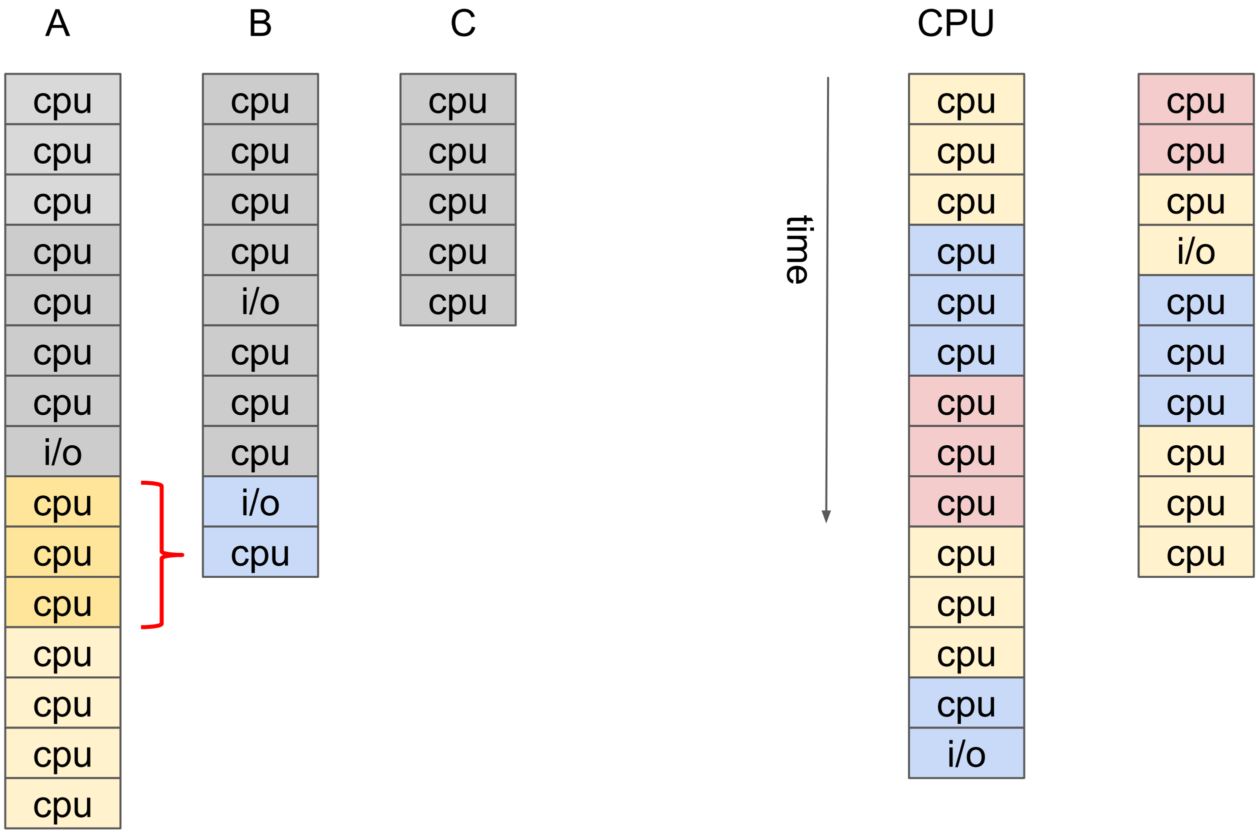
Exercise


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
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
i/o
cpu
cpu
cpu
i/o

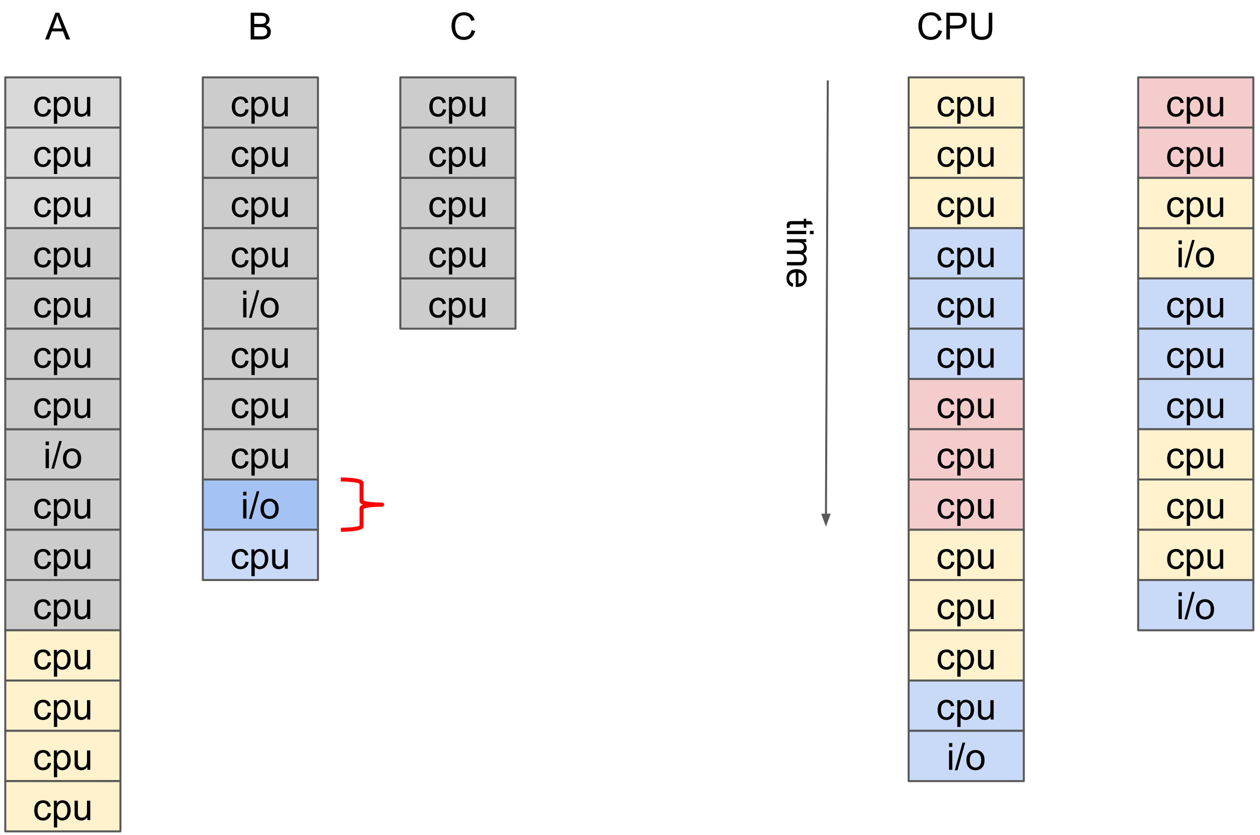
Exercise



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, cpu, cpu, cpu, cpu, cpu, cpu, cpu, i/o
cpu, cpu, cpu, i/o, cpu, cpu, cpu, cpu, cpu, cpu

Exercise

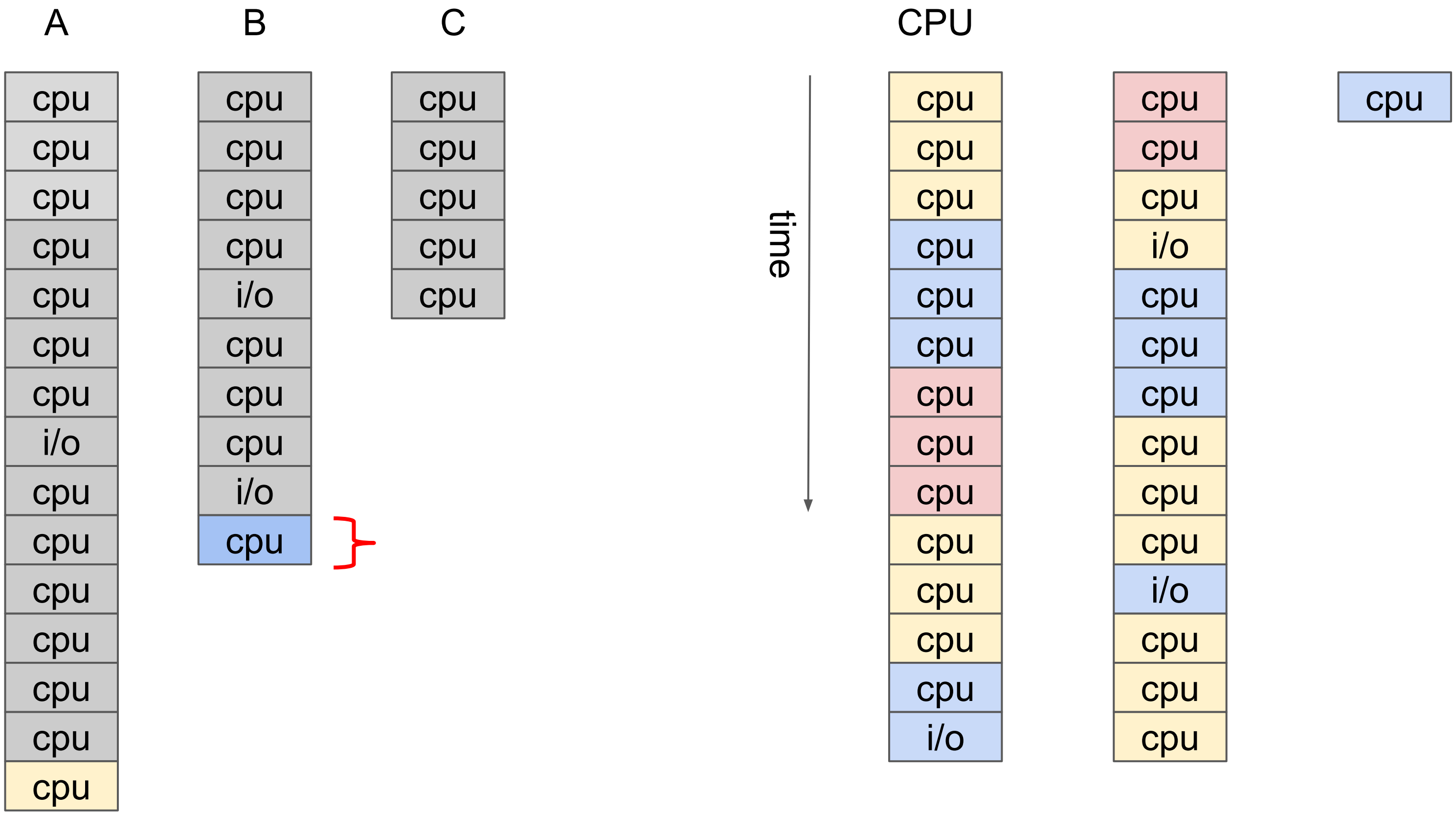

A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
i/o

Exercise

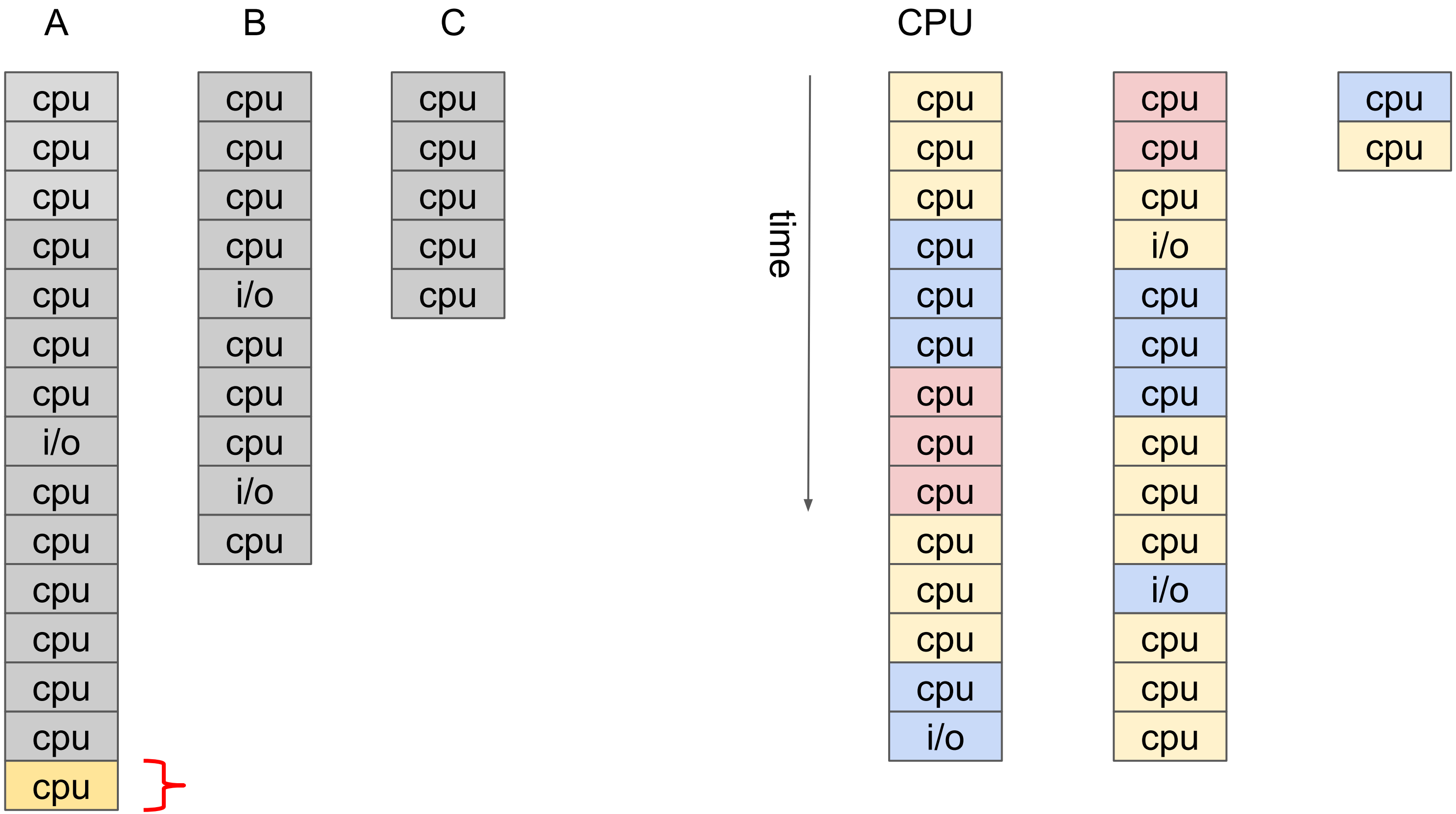
time

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

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 i/o
cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu
cpu

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
CPU
time
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 switch

- context switch is an essential part of any multitasking OS
- we need context switching to implement multitasking when # CPUs < # processes
- allows (the illusion of) sharing of a small number of CPUs among many processes

- OS maintains a context (state) for each process
  - usually part of PCB, includes saved registers, open files, ...
- when OS switches between processes A and B:
  - OS first saves A's state in A's PCB
    - eg. save current CPU registers into PCB_A
  - OS then restores B's state from B's PCB
    - eg. load CPU registers from PCB_B

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 could be improved by hardware support:
  - eg. some CPUs support saving/restoring 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

- a form of **interprocess communication (IPC)**
- similar concept to hardware interrupts on CPUs (you can think of it as process interrupt)
- signals are asynchronous
- very limited form of IPC - 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 via **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

eg. on segmentation fault kernel sends `SIGSEGV`

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

- from command line

```
$ kill 12345    # tries to kill a specific process with signal SIGTERM
$ kill -9 12345 # kills a specific process with signal 9 → SIGKILL
$ kill -9 -1    # kills all processes except pid=1 (init)
```

- 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 '**caught signals**'
- not all signals can be **caught**
    - `$ 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(2)` 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
  - also signal handler could itself be interrupted !!!
- when writing a signal handler:
  - keep it simple, for example:
  - modify a global flag and let the program handle the interrupt later
  - declare global variables with volatile keyword (eg. `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

Reentrant functions

- reentrant functions are functions:
  - that can be interrupted in the middle of an operation
  - and then called again (re-entered)
  - and finally the original function call can finish executing
- used in interrupt handlers, signal handlers, multi-threaded applications, recursion *
- when writing reentrant functions:
  - don't use global variables
    - there are some exceptions, eg. using atomic operations
  - do not call non-reentrant functions
    - unless you can temporarily disable interrupts / signals

Reentrant 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?

Reentrant functions

- example of a reentrant 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 reentrant 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

```
#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> <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);
}
```

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.
```

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++

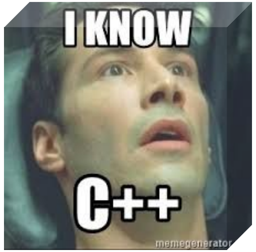
  https://en.cppreference.com/w/cpp/utility/program/signal

swap() in C and C++

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

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

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

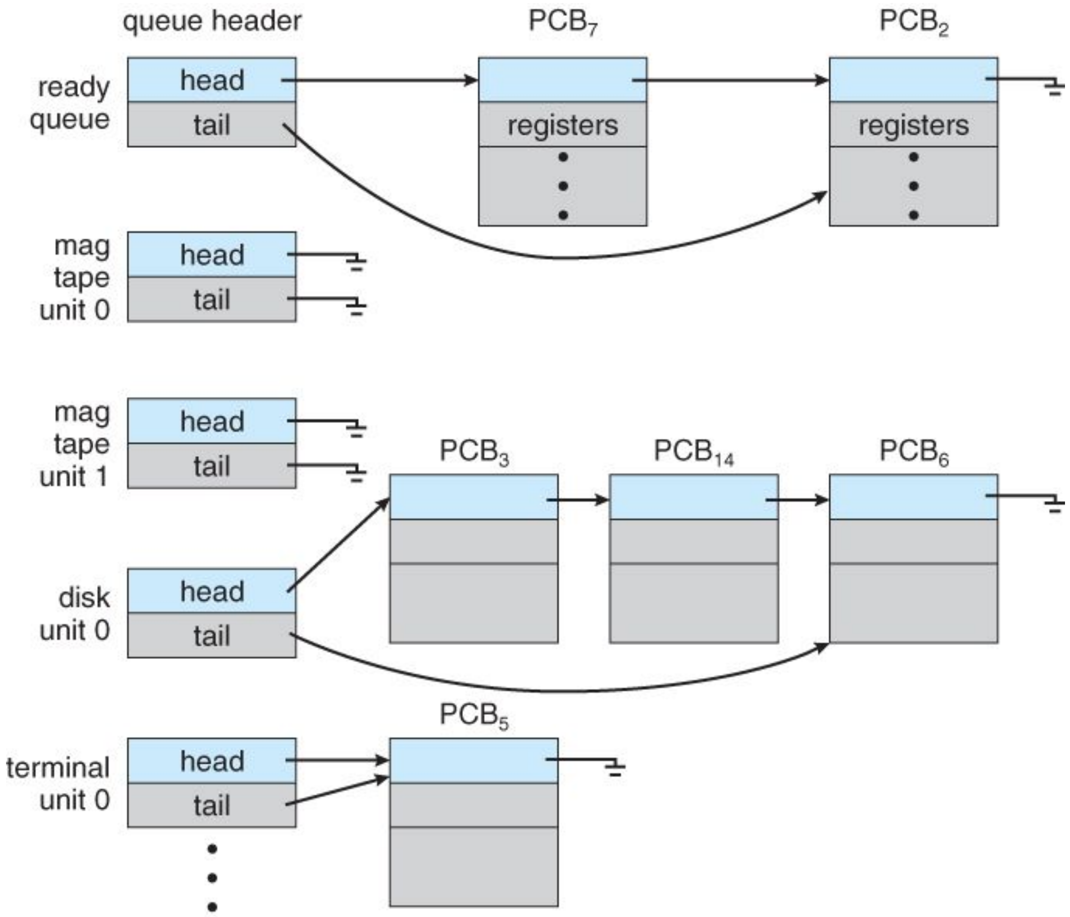
```cpp
// swap in C++
// references are cool

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);
```



Questions?

Queues



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