Scheduling Algorithms, I

Usual: simple algorithm with "hooks"

Want simple algorithm so that it runs **fast**

Typical:

- Round-robin variants – e.g. multiple priority levels; round-robin within a level; some way to move schedulable unit (SU) up/down among priorities
- Lottery scheduling – each ready-to-run SU gets some number of "tickets" then a ticket is chosen at random

Scheduling Algorithms, II

Many OSes have “hooks” that allow user code to request ...

- **Processor affinity**: SU will be scheduled on a particular processor
- **Gang scheduling**: N SUs want to run together simultaneously; therefore, none is scheduled unless all N can be placed on N different processors simultaneously

For this course: **we assume OS scheduling algorithm is unknown and uncontrollable; ANY sequence of actions is possible**

The Process, III

Process consists of:

- Process ID—unchanging, unique ID to distinguish it from all other processes
- Private *virtual address space*—set of memory locations process may access, and how it may access them (read-only, read/write, maybe some other possibilities)
- Instructions—from one or more executable files—placed into address space
- At least one **thread** of execution—unit of scheduling
- Set of “resources” (see below)

Process Resources

OS resources associated with process:

- Uid/gid, euid/egid, parent pid
- Current directory, current root
- Open files
- Signal state (which are being handled, which pending)
- Process group & current terminal/window (UNIX/Windows, respectively)
- Accounting info & scheduling parameters (e.g., priority, CPU time consumed)
- Address space mappings: indications of text, data, and stack segments

Thread Resources

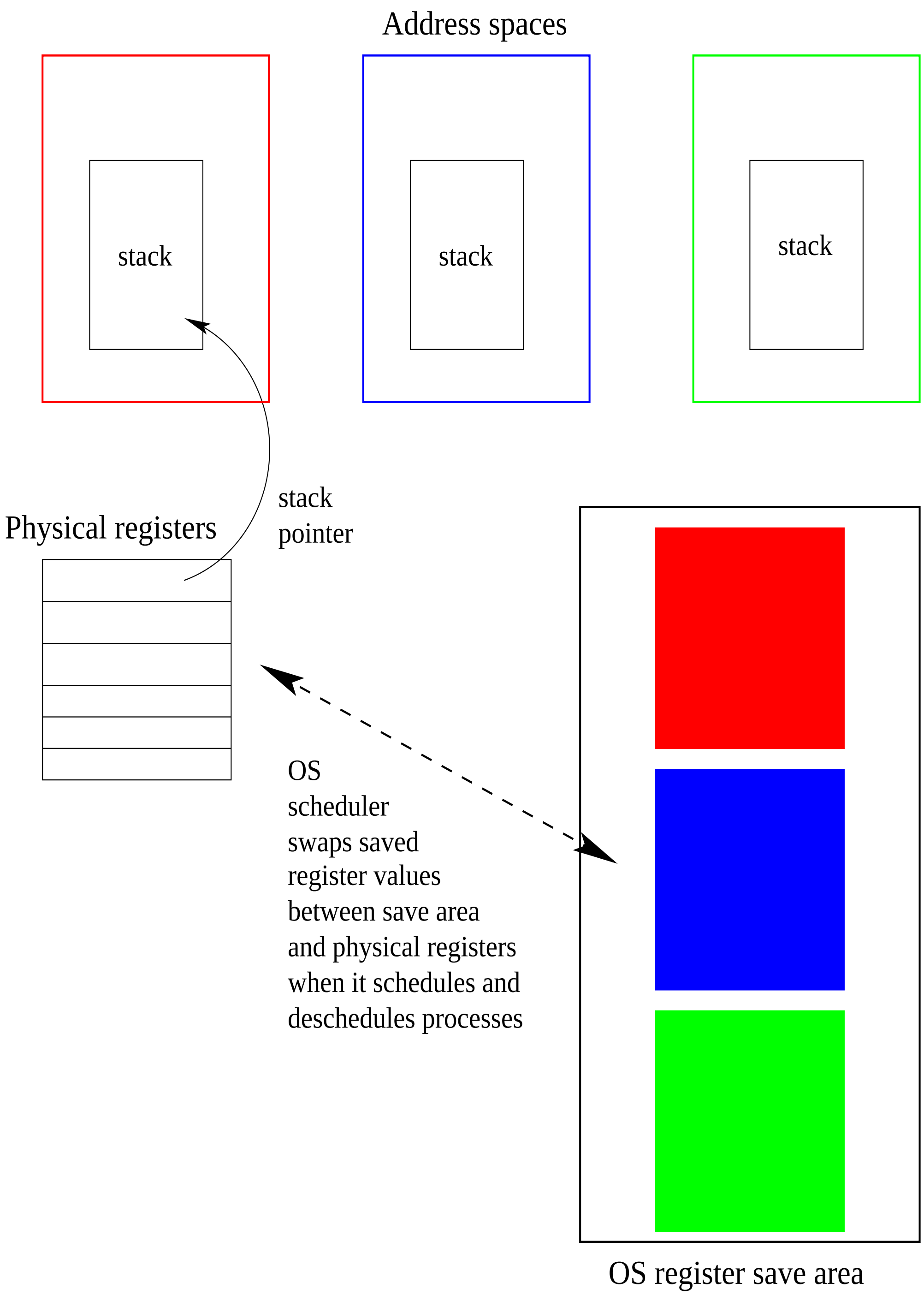
Resources associated with thread:

- Registers (incl. program counter and stack pointer)
- Stack
- PSW (includes processor interrupt priority)

These are all hardware resources

Generally, OS abstract resources are associated w/ process & shared by all threads

Register & Stack Swap



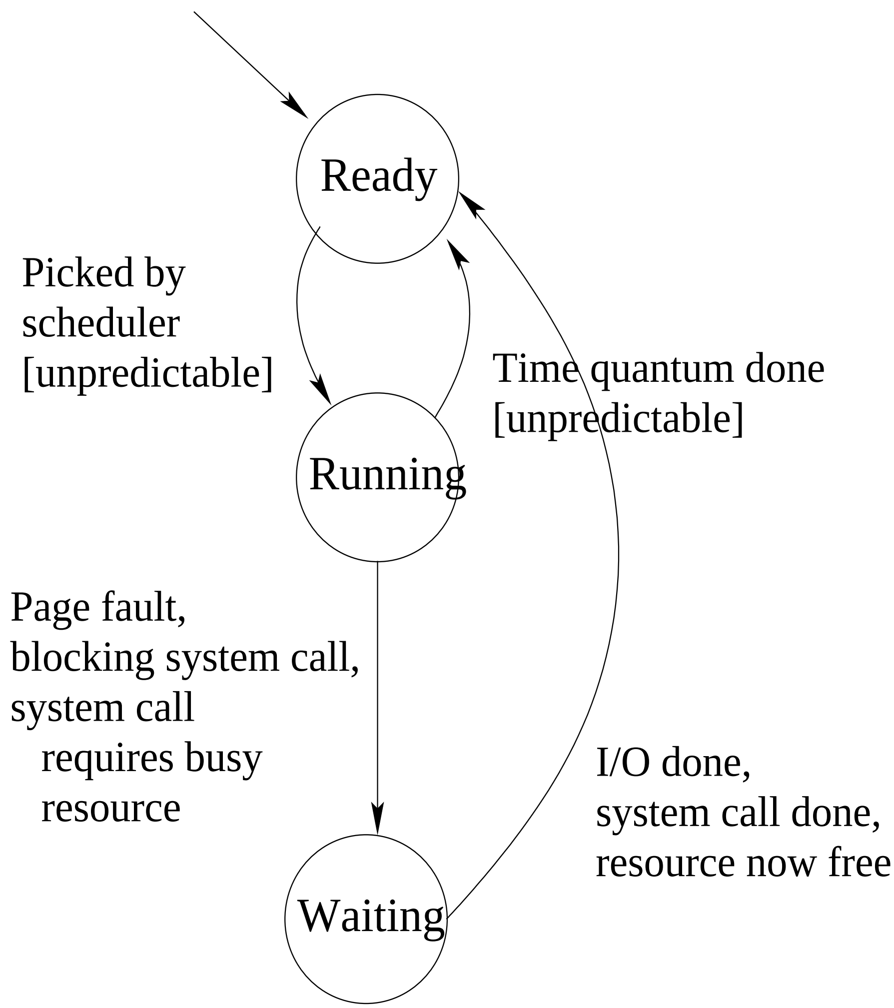
Thread States

Three major states:

1. Running (only one at a time on uniprocessor)
2. Ready
3. Waiting (for some resource)

"Waiting" is an abstraction—there are MANY wait states, one for each resource type

State Transitions



Process Creation, I

`fork()` makes near-exact copy of running process

`exec()` makes existing process run specified executable file from the beginning

`exec()` is a **loader**

Use of `fork()` and `exec()`:

```
pid_t child;
if ((child = fork()) < 0) {
   // ERROR:
   // errno indicates which error
} else if (child == 0) {
   // CHILD:
   // fork returns 0 to child
   // child typically calls exec soon
} else {
   // PARENT:
   // fork returns child's pid to parent
   // parent knows child pid, but not its own
}
```

Example: forkexample.c

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

int main()
{
  int pid;

  pid = fork();

  if (pid < 0) {
    fprintf(stderr, "Fork failed\n");
    exit(-1);
  }
  else if (pid==0) {    /* child process */
    printf("Child: pid=%d\n", pid);
    execlp("/bin/ps", "ps", NULL);
  }
  else {                /* parent process */
    printf("Parent: child pid=%d\n", pid);
    printf("Parent exiting!\n");
    exit(0);
  }

  return 0;
}
```

Process Creation, II

First few processes are specially created during OS initialization

Process 0, `swapper`, is the scheduler

Process 1, `/sbin/init`, processes `/etc/rc` files

Process 2, `pagedaemon`, plays role in virtual memory

`swapper` and `pagedaemon` are **kernel processes**—no executable file, not created by `fork`