Systemy operacyjne

Wykład 6: Buforowanie

Koszt wywołań systemowych

Na podstawie *“The Linux Programming Interface”*:

1. Jądro Linux 2.6.30
2. System plików: **`ext2`**
3. Rozmiar bloku systemu plików: 4096 bajtów
4. Bufor w przestrzeni użytkownika: **`BUF_SIZE`** bajtów
5. Rozmiar pliku: ~100M bajtów

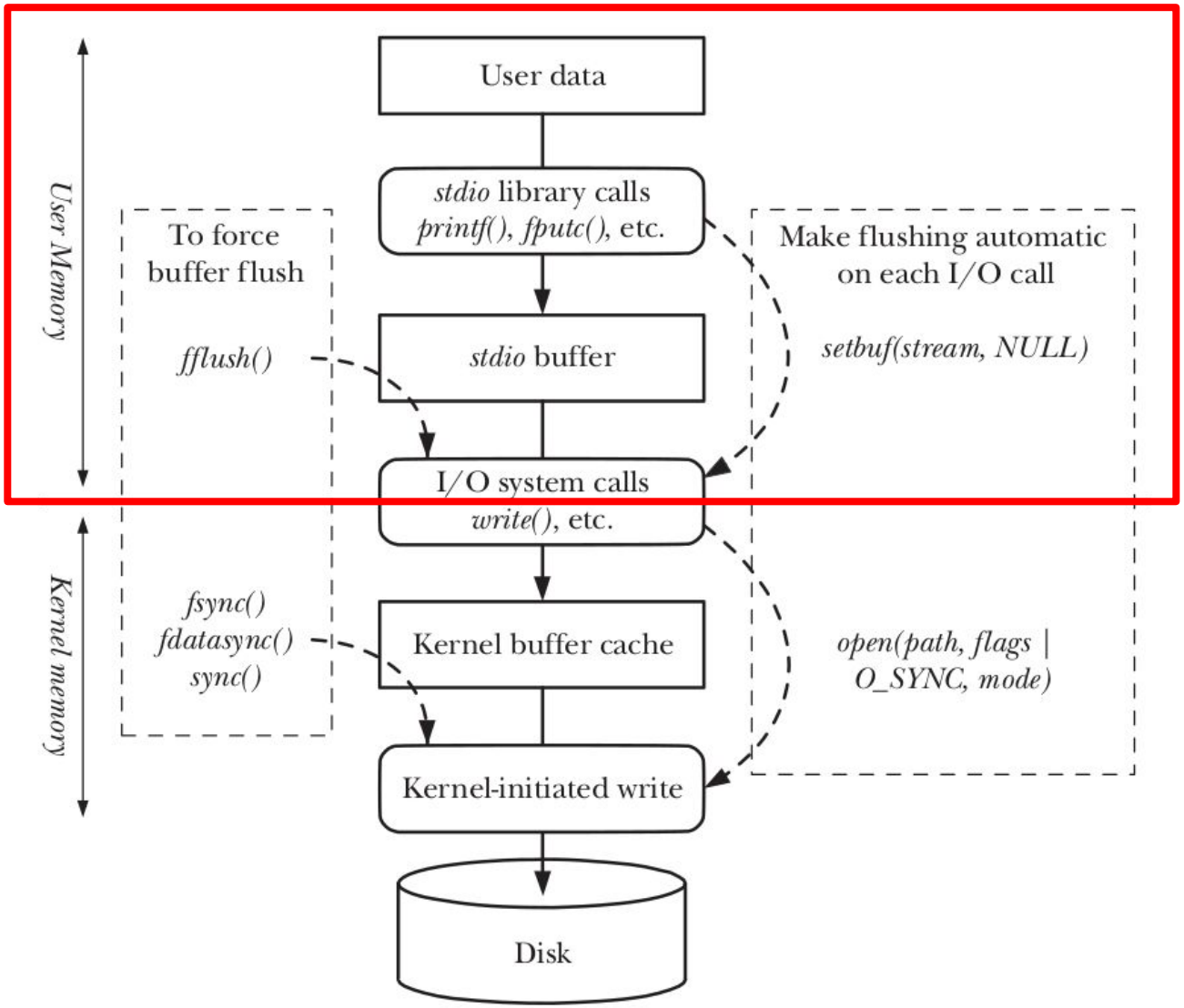
Kopiowanie pliku z użyciem `read(2)` i `write(2)`

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Tworzenie zawartości pliku z użyciem `write(2)`

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	72.13	72.11	5.00	67.11
2	36.19	36.17	2.47	33.70
4	20.01	19.99	1.26	18.73
8	9.35	9.32	0.62	8.70
16	4.70	4.68	0.31	4.37
32	2.39	2.39	0.16	2.23
64	1.24	1.24	0.07	1.16
128	0.67	0.67	0.04	0.63
256	0.38	0.38	0.02	0.36
512	0.24	0.24	0.01	0.23
1024	0.17	0.17	0.01	0.16
4096	0.11	0.11	0.00	0.11
16384	0.10	0.10	0.00	0.10
65536	0.09	0.09	0.00	0.09

Buforowanie plików w przestrzeni użytkownika



Buforowanie biblioteki **`stdio`**

W strukturze **`FILE`** każdego strumienia jeden bufor.

```
void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf,
            int mode, size_t size);
```

Function	*mode*	*buf*	Buffer and length	Type of buffering
`setbuf`		non-null	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	non-null	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	non-null	user *buf* of length *size*	line buffered
		`NULL`	system buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

Domyślne tryb buforowania

Bufor domyślnie opróżniany w trakcie zamykania pliku **`fclose(3)`** otwartego do zapisu. W trakcie pracy możemy zawołać **`fflush(3)`**, żeby jawnie go opróżnić.

Dla plików dyskowych buforowanie pełne, dla plików terminala buforowanie liniami, dla **`stderr`** brak buforowania.

Jak sprawdzić czy plik jest terminalem?

```
int fileno(FILE *stream);
int isatty(int fd);
```

Domyślna wielkość bufora → **`st_blksize`** (**`statbuf`**).

Problemy z buforowaniem po stronie użytkownika

1. Podwójne kopiowanie danych:
   - jądro kopiuje dane do bufora FILE,
   - użytkownik korzystając z funkcji **`stdio`** kopiuje dane do własnej pamięci.
2. Utrata zawartości bufora:
   - zapisujemy dane z użyciem **`fprintf(3)`** albo **`fwrite(3)`**
   - zapominamy zawołać **`fclose(3)`**
   - wychodzimy z programu…
   - albo przychodzi sygnał, który kończy działanie programu.

Wywołania systemowe **`readv(2)`** i **`writev(2)`**

Motywacja: Piszemy nasz własny edytor tekstu.
Plik reprezentujemy w pamięci jako tablicę rekordów:

(długość linii, wskaźnik do zawartości linii)

Linie nie muszą być ułożone w pamięci sekwencyjnie!

Co musimy zrobić, żeby zapisać plik na dysk?

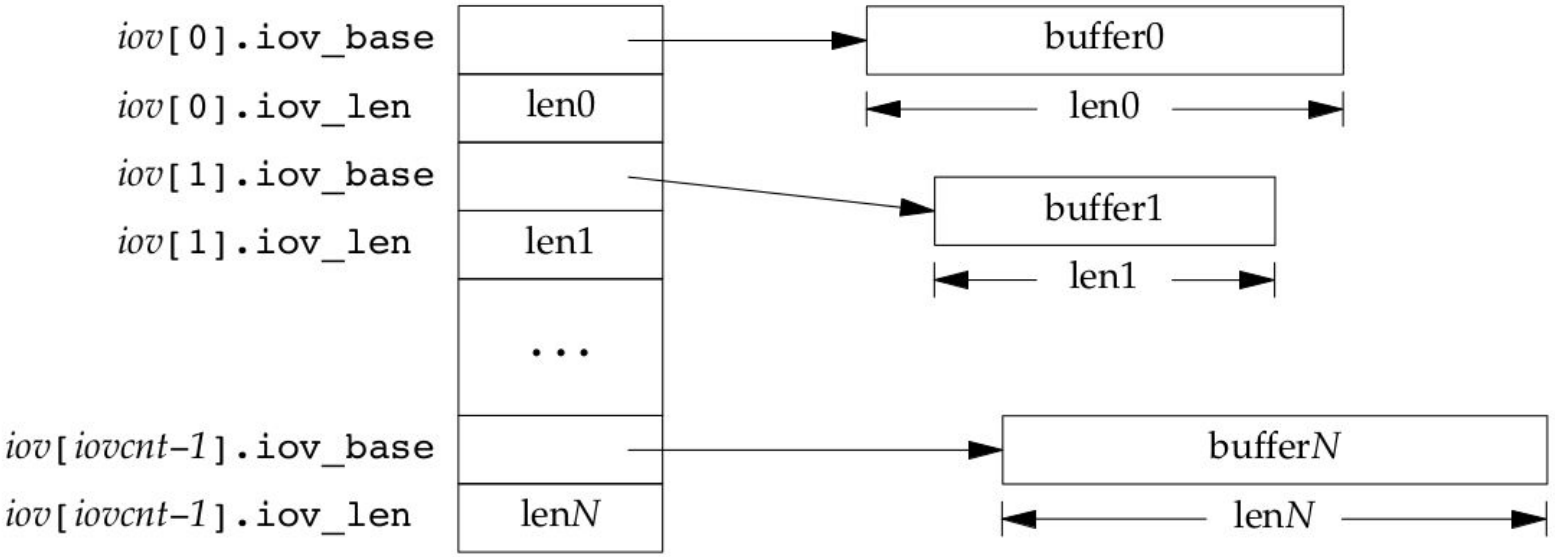
1. Dla każdej linii zwołać write(2).
2. Przygotować jeden wielki bufor, do którego wkopiujemy wszystkie linie i zapiszemy na dysk w jednym kroku.

Wywołania systemowe **`readv(2)`** i **`writev(2)`** c.d.

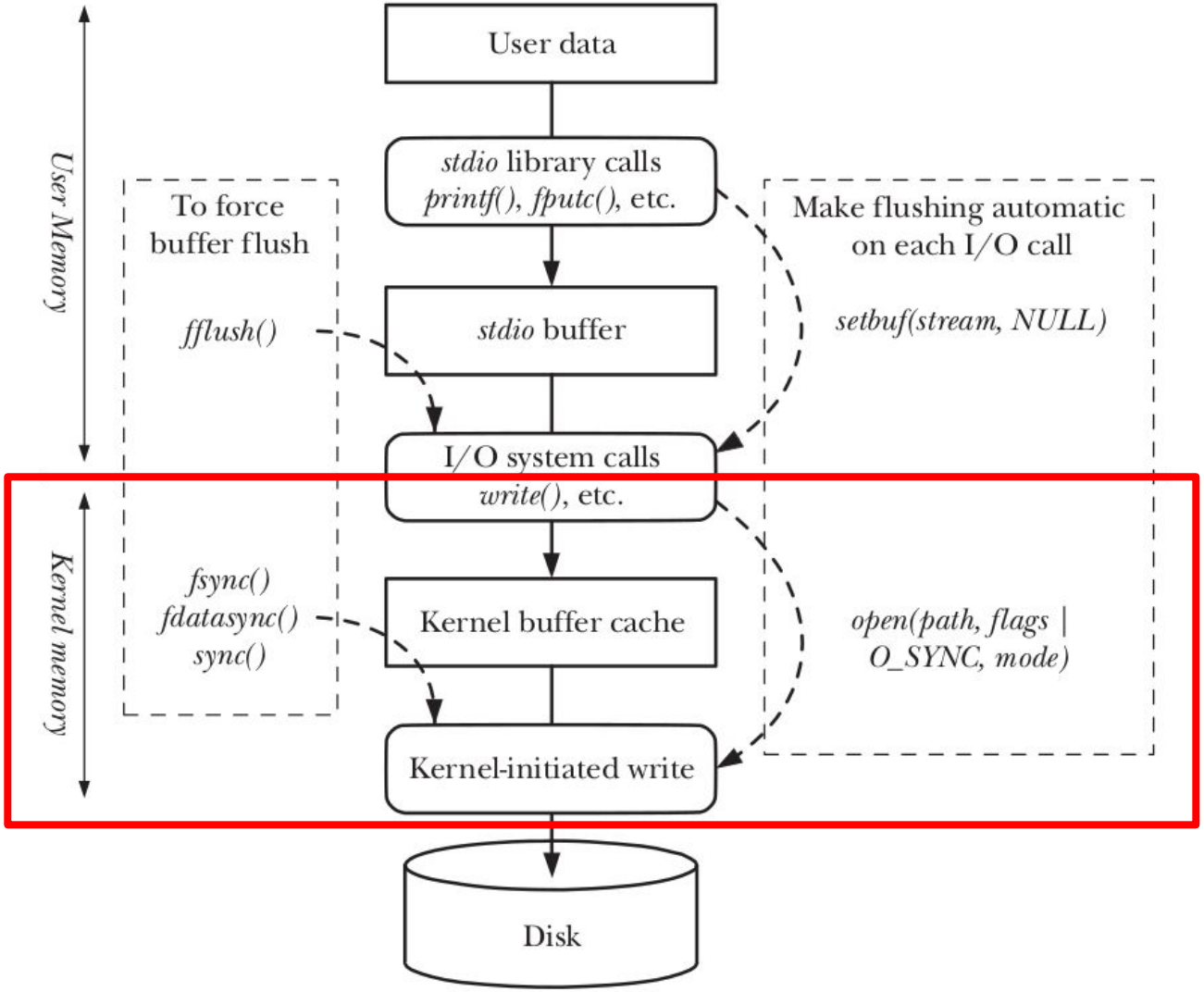
Rozwiązaniem *scatter read* i *gather write*:

```
ssize_t readv(int fd, struct iovec *iov, int iovcnt);

ssize_t writev(int fd, struct iovec *iov, int iovcnt);
```

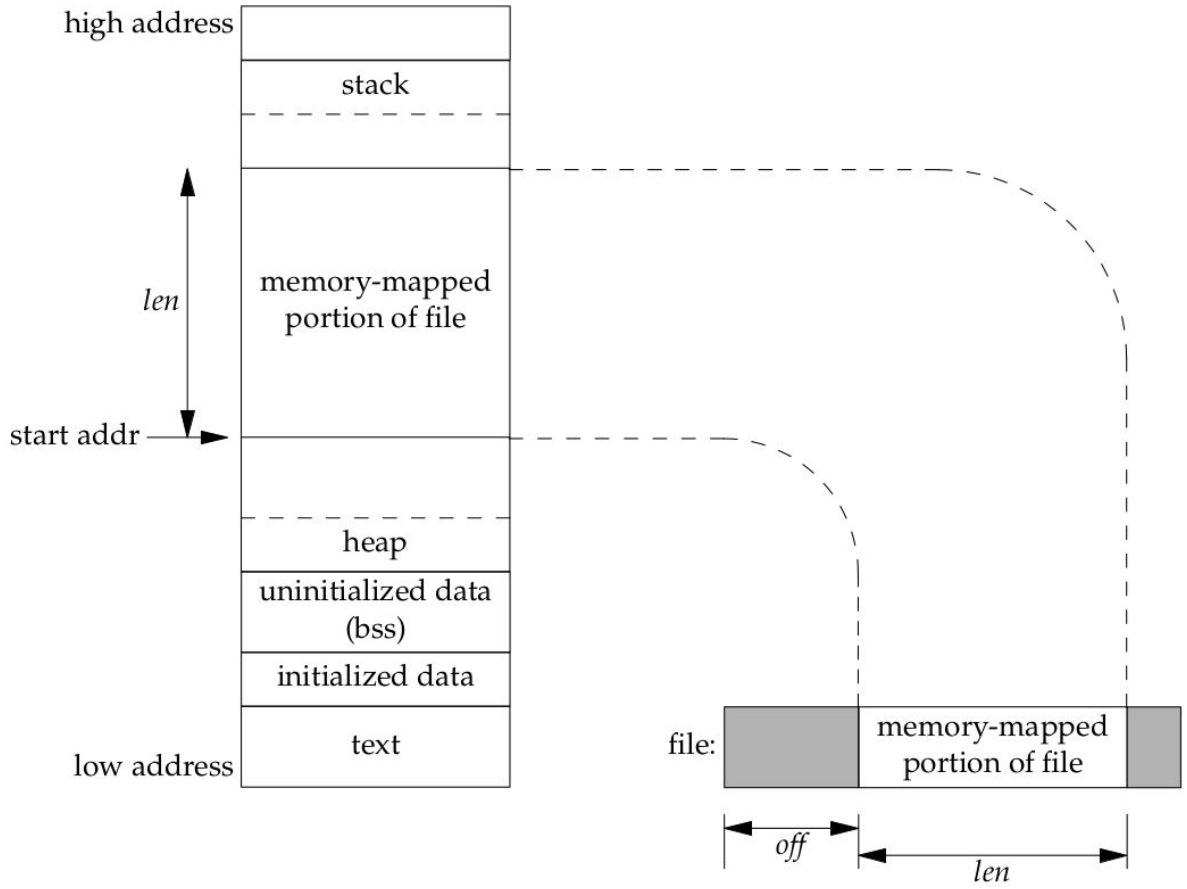


Buforowanie plików w przestrzeni użytkownika



Pliki odwzorowane w pamięć

```
void *mmap(void *addr, size_t len, int prot, int flags,
           int fd, off_t off);
```



Jak system buforuje dane?

1. W procesie A otwieramy plik F z flagą **`O_RDWR`**.
2. W procesie B tworzymy odwzorowanie współdzielone **`MAP_SHARED`** całego pliku F na pamięć.

Kilka kłopotliwych pytań:

- Czy jeśli A zapisze do F z użyciem **`write(2)`** to B zobaczy zmiany w swojej pamięci?
- Czy jeśli B zapisze pod adresy, w które odwzorowano plik F, a bezpośrednio przed kursor F w A, to operacja **`read(2)`** wczyta świeże dane?

Czy widok A i B na zawartość F mogą być różne?

Pamięć podręczna buforów

Jądro utrzymuje sprowadzone z dysku bloki w ***buffer cache*** (termin stosowany zamiennie z ***page cache***).

Bufory te mogą być bezpośrednio odwzorowane w pamięci użytkownika (**`mmap`**), albo używane przez implementację wywołań systemowych **`read`** i **`write`**.

Zapis do pliku kończy się jedynie zapisem do bufora.
Po jakimś czasie jądro zapisze zawartość buforów na dysk.

Jądro wykorzystuje wolną pamięć RAM do cache'owania pamięci drugorzędnej (nośniki danych).

Problem z buforowaniem

Rozważmy serwer poczty przekazujący e-mail do serwera B:

1. Odbiera e-mail z serwera A
2. Zapisuje go na dysku
3. Wysyła potwierdzenie (odebrałem!) do serwera A
4. Serwer A kasuje wiadomość z dysku
5. Serwer B przekazuje e-mail dalej

Jeśli serwer B ulegnie awarii (np. brak prądu) po wykonaniu punktu 3, to czy e-mail będzie bezpieczny?

Nie! Zawartość pliku z wiadomością mogła być nadal przechowywana w buforze systemu plików (RAM).

Buforowanie danych i metadanych

- **`sync(2)`** synchronizuje wszystkie bufory jądra z pamięcią drugorzędną
- **`fsync(2)`** synchronizuje dane i metadane wybranego pliku
- **`fdatasync(2)`** synchronizuje tylko dane pliku

Jaka jest różnica? Dopisujemy na koniec pliku – dane zostały wypisane na dysk, a rozmiar pliku nie, bo jest w metadanych!

Dodatkowo w trakcie otwierania pliku możemy przekazać do **`open(2)`** flagi **`O_SYNC`** i **`O_DSYNC`**, które mają taki sam efekt co **`fsync`** i **`fdatasync`** przy każdej operacji **`write`**.

Pytanie: Czy **`*sync`** zapewnia spójności danych na dysku?

Pytania?